EINDHOVEN UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF MATHEMATICS AND COMPUTING SCIENCE

MASTER'S THESIS

FROM S-ALGEBRAS TO TEMPLATE CLASSES: A STRUCTURE EDITOR WITH MULTIPLE VIEWS

BY

HUGO M.H. LYPPENS

SUPERVISOR:	DR. IR. C. HEMERIK
ADVISOR:	B.W. WATSON

APRIL, 1995

Contents

1 Introduction

This thesis describes the design and implementation of a *multiple-view structure editor*. The conventional way to enter a program or any language element is to type a textual representation, using a text editor. If it is well-formed, a parser can generate the corresponding syntax tree structure from it. On the other hand, a structure editor is syntax-aware and allows direct manipulation of the language element's syntax tree.

This report first presents a mathematical framework, and shows how this framework is used as a guide in the structure editor design process. An important design aim is to make it possible to have different *views*. Each view has its own way of presenting and letting a user manipulate the structure. Many views can be used simultaneously, and if the user manipulates one of them, the others should be updated accordingly.

The multiple-view structure editor presented in this report is based on Mr. Hemerik's structure editor, written using PASCAL with Objects and TurboVision. It will be investigated how a different implementation environment affects the structure editor design. The structure editor presented here is targeted at the MS-Windows platform. It is implemented in C++, using the ObjectWindows Library as object-oriented application framework and interface to MS-Windows.

2 Mathematical Modelling and Problem Formulation

A structure editor relies on a language description based on *abstract syntax*. This chapter explains signatures and related mathematical concepts which are used to model the structure to be edited.

There is an important distinction between ‘abstract’ and ‘concrete’ syntax. A *grammar* is a method to specify the concrete syntax of a language. It involves four quantities: *terminals, nonterminals* (also called *syntactic categories*), a *start symbol* and *production rules.*

A common way to describe a grammar is known as BNF, which stands for ‘Backus-Naur Form’. In the following, we will use the *assignment*, the *if-then-else* conditional construct, the *while* repetition construct and the *statement-list* construct (used to combine a sequence of statements to form a single statement) that appear in many programming languages as a running example. Using BNF, the definition of these constructs in PASCAL would look like:

```
Expr        ::= True | False | Identifier
Identifier  ::= a | b | ... | z
Stat        ::= Identifier := Expr
Stat        ::= if Expr then Stat else Stat
Stat        ::= while Expr do Stat
Stat        ::= begin Stat-list end
Stat-list   ::= Stat | Stat ; Stat-list
```

In these production rules, words in bold type are terminals and ‘Stat’, ‘Stat-list’, ‘Identifier’ and ‘Expr’ are nonterminals. The extra nonterminal ‘Stat-list’ is needed, because BNF requires each production rule to have a known, fixed number of terminals and nonterminals at the right-hand side. BNF describes the external appearance of programs, since its production rules include keywords and other external syntactic conventions; it gives the *concrete syntax* of a language.

In contrast, an *abstract syntax* description leaves out keywords and other details and simply specifies the components of each language construct. If a language has the *if-then-else* construct, it would contain a way to form a statement from a Boolean expression and two other statements (one to be executed when the Boolean expression evaluates to TRUE and another to be executed when it evaluates to FALSE). The advantage of the abstract syntax description of a language is that it directly corresponds to its algebraic structure. For a program or, more generally speaking, a term, an abstract syntax tree is a far more fundamental representation than its concrete form. To go from abstract syntax tree to concrete representation, one only needs to insert *delimiters/decorations* (the underlined entities in the above BNF definition) at the correct places. Going the other way around is more difficult: this amounts to the parsing problem. Therefore, the structure editor developed uses an abstract syntax tree representation of the program or term to be edited.

2.1 Basics

This paragraph gives the definitions of basic mathematical constructs used throughout this thesis.

Definition 2.1

Assume I to be any nonempty set and $k \in N$.

A **sequence** *s* of length *k* over I is a function of type $\{i \in N \mid i < k\} \to I$.

$I^k := \{i \in N \mid i < k\} \to I$, the set of sequences of length *k* over *I*.

$I^* := \{\cup i \colon i \in N : I^i\}$, the set of sequences over *I*.

e is the empty sequence. $I^0 = \{$e$\}$

| s | is the **length** of a sequence s.

The following abbreviation can be used to construct sequence of length *k* over *I* from given values $v_0, \ldots, v_{k-1} \in I$:

$\langle v_0, \ldots, v_{k-1} \rangle := \{(0, v_0), \ldots, (k-1, v_{k-1})\}$

To test whether a sequence *r* over *I* is a prefix of a sequence *s* over *I*, an operator $\leq_p : I^* \times I^* \to Boolean$ is used:

$r \leq_p s := r \subseteq s$ or, more verbose, $r \leq_p s := (\exists t \in I^* :: r \oplus t = s)$

□

Definition 2.2

To construct a sequence from the leftmost *k* (valid for any k Î N) elements of an existing sequence *s* over *I*, we use the left-take operator $lt \colon I^*$ ´ N ® I^* .

$lt(s,k)$: **=** $\{(i, s(i)) \mid 0$ **£** i **<** $\min(k, |s|)\}$

To construct a new sequence by dropping the leftmost $k \in N$ elements from an existing sequence *s* over *I*, we use the left-drop operator $ld \colon I^*$ ´ N ® I^* .

$ld(s,k) := \{(i, s(i+k)) \mid 0 \leq i \lhd |s| - k\}$

To construct a sequence from the rightmost $k \in N$ elements of an existing sequence *s* over *I*, we use the right-take operator $rt \colon I^* \times N \to I^*$.

$rt(s,k) := ld(s, |s| - k)$

Finally, to construct a new sequence by dropping the rightmost $k \in N$ elements from an existing sequence *s* over *I*, we use the right-drop operator $rd \colon I^* \times N \to I^*$.

$rd(s,k) := lt(s, |s| - k)$

□

Definition 2.3

Assume I to be any nonempty set.

The concatenation of two sequences over I is the result of the -infix denoted- function $\oplus : I^* \times I^* \to I^*$.

Given $l_0, l_1 \in N, s_0 \in I^{l_0}, s_1 \in I^{l_1}$, then

$s_0 \oplus s_1 := s_0 \cup \{i : i \in N \wedge i < l_1 : (l_0 + i, s_1(i))\}$

Obviously, $s_0 \oplus s_1 \in I^{l_0 + l_1}$.

□

Definition 2.4

Assume I to be any nonempty set.

To insert an element $e \in I$ into a sequence s over I, at location $n \in N, n \leq |s|$, a function $insert : I^* \times N \times I \to I^*$ is used.

$insert(s, n, e) := lt(s, n) \oplus < e > \oplus ld(s, n)$

Obviously, $|insert(s, n, e)| = |s| + 1$.

□

Definition 2.5

Assume I to be any nonempty set.

To delete from a sequence s over I element n, $n \in N \wedge n < |s|$, a function $delete : I^* \times N \to I^*$ is used.

$delete(s, n) := lt(s, n) \oplus ld(s, n + 1)$

Obviously, $|delete(s, n)| = |s| - 1$.

□

Definition 2.6

Assuming S to be any set (the elements of which are called *sorts*), an **S-sorted set** X is an S indexed set of (possibly empty) sets $X_s, s \in S$ such that:

- $X = \{X_s | s \in S\}$
- $\forall s, t \in S : s \neq t : X_s \cap X_t = \varnothing$

If S is nonempty, then the element of X corresponding to an $s \in S$ is written as X_s.

The following abbreviation is used to construct the union of all elements of S-sorted set X:

$\cup X := (\cup s : s \in S : X_s)$.

□

2.2 Signatures

In this paper, a *signature* is used to define the abstract syntax of the structure that will be edited.

Definition 2.7

A **signature with list operators and S** is an extension of the standard signature concept. It is a pair (S, **G**), where S is a nonempty, finite set of *sorts* and **G** is a finite set of sets of *operators*. The set **G** is $S^* \times S \cup S \times S$ -sorted: its elements are the following pairwise disjoint sets:

- $\Gamma_{<a,r>}$, the (possibly empty) set of operators with a fixed number of arguments (aggregate operators) with argument sort sequence $a \in S^*$ and result sort $r \in S$.
- $\Gamma_{<s,r>}$, the (possibly empty) set of operators with a variable number of arguments (list operators) with arguments all of the same sort $s \in S$ and result sort $t \in S$.

for all $a \in S^*, r \in S, s \in S, t \in S$.

◻

Given a signature **S** (S, **G**) and a sort $r \in S$ we may use the following shorthand to refer to all operators having result sort *r*:

$$\Gamma_r := \left(\cup s{:}\, s \in S^* : \Gamma_{<s,r>}\right) \cup \left(\cup s{:}\, s \in S : \Gamma_{<s,r>}\right)$$

A *sort* can be thought of as the name of a syntactic category. Examples are ‘Expr’ and ‘Stat’.

An *operator* can be seen as a production rule; its name often corresponds to a language construct. An aggregate operator’s *arity* is defined as its number of arguments.

- A constant, defined as $g{:} \rightarrow s_0$, is an aggregate operator of arity 0. Example: $True{:} \rightarrow Expr$.
- An aggregate operator has a fixed number of arguments, each of a certain sort (constants are a special kind of aggregate operator). We distinguish two ways to define an aggregate operator *g* of arity *n*: unlabeled and labeled.

  Unlabeled: $g{:} s_1 \times \ldots \times s_n \rightarrow s_0$. For example, the definition for the abovementioned if-then-else is: $ifelse{:} Expr \times Stat \times Stat \rightarrow Stat$.

  Labeled: $g{:} \{ f_1 : s_1 ,\ldots, f_n : s_n \} \rightarrow s_0$, where the names f_1 to f_n are labels to distinguish the arguments. This definition resembles the definition of a *record type* in programming languages. A possible labeled if-then-else definition is:

  $ifelse{:}\{guard{:}\ Expr, s\_true{:}\ Stat, s\_\ false{:}\ Stat\} \rightarrow Stat$.
- A list operator has a variable number of arguments, each of the same sort. Again, the operator definition can be given in unlabeled and labeled forms.

  Unlabeled: $g{:} s_1^* \rightarrow s_0$. An example of this is $statlist{:} Stat^* \rightarrow Stat$, the statement list construct. As usual, the asterisk indicates zero or more repetitions.

  Labeled: $g{:} \{ f_1 {:} s_1^* \} \rightarrow s_0$. A labeled statement list construct: $statlist{:} \{s{:} Stat^*\} \rightarrow Stat$.

Depending on the actual discussion we can freely choose the unlabeled or labeled form of the operator definitions (see also [1], Section 3.4). In both forms, we can refer to the operator’s arguments by their ordinal number. In case of an aggregate operator, this number ranges from 0 to (operator arity)-1 and in case of a list operator, it ranges from 0 to (argument sequence length)-1.

Given a signature **S** (S, **G**) and an operator **g**, the predicate $IsListOp{:} \cup\Gamma \rightarrow Boolean$ can be used to find out if this operator is a list operator:

$$IsListOp(\mathrm{g}) := \left(\exists s, r : s \in S, r \in S : \mathrm{g} \in \Gamma_{<s,r>}\right)$$

The full, unlabeled signature for our running example is:

({Stat, Expr, Identifier}, {ifelse, while, statlist, assignment, a, b, ..., z, True, False, value})

$$\begin{aligned}
&assignment: Identifier \times Expr \to Stat\\
&ifelse: Expr \times Stat \times Stat \to Stat\\
&while: Expr \times Stat \to Stat\\
&statlist: Stat^{*} \to Stat\\
&a, b, \ldots, z: \to Identifier\\
&value: Identifier \to Expr\\
&False: \to Expr\\
&True: \to Expr
\end{aligned}$$

And this is a labeled version:

({Stat, Expr, Identifier}, {ifelse, while, statlist, assignment, a, b, ..., z, True, False, value})

$$\begin{aligned}
&assignment: \{dest: Identifier, source: Expr\} \to Stat\\
&ifelse: \{guard: Expr, s\_true: Stat, s\_false: Stat\} \to Stat\\
&while: \{guard: Expr, body: Stat\} \to Stat\\
&statlist: \{s: Stat^{*}\} \to Stat\\
&a, b, \ldots, z: \to Identifier\\
&value: \{source: Identifier\} \to Expr\\
&False: \to Expr\\
&True: \to Expr
\end{aligned}$$

2.3 Terms over a signature

The following definition concerns the set of terms that can be derived using a signature. See also: [3], Chapter 3.

Definition 2.8

Given a signature **S** = (S, **G**), the S-sorted set $Term_{\Sigma}$ is the smallest S-sorted set such that

$$\text{"}n \;\hat{I}\; N :: \begin{array}{l} ((\text{"}g, a \;\hat{I}\; S^{n}, r \;\hat{I}\; S : g \;\hat{I}\; \mathbf{G}_{<a,r>} : (\text{"}i : i \;\hat{I}\; N \;\grave{U}\; i < n : t_i \;\hat{I}\; Term_{\mathbf{S}_{a(i)}}) \;\text{Þ}\; g[t_0, \ldots, t_{n-1}] \;\hat{I}\; Term_{\mathbf{S}_r}) \;\grave{U} \\ (\text{"}g, s \;\hat{I}\; S, r \;\hat{I}\; S : g \;\hat{I}\; \mathbf{G}_{<s,r>} : (\text{"}i : i \;\hat{I}\; N \;\grave{U}\; i < n : t_i \;\hat{I}\; Term_{\mathbf{S}_s}) \;\text{Þ}\; g[< t_0, \ldots, t_{n-1} >] \;\hat{I}\; Term_{\mathbf{S}_r})) \end{array} .$$

The square brackets are used syntactically here. We adopt the convention that $\mathrm{g}[]$ is simply written as g.

□

In language terms: the set $Term_{\Sigma}$ is the abstract language generated from the abstract syntax modelled by **S**.

An example term (element of $Term_{\Sigma_{Stat}}$) available through the signature **S** for our running example:

ifelse[value[a],assignment[b,False],assignment[c,True]]

2.4 S-algebra

Definition 2.9

A **S-algebra** with respect to a signature with list operators **S** = (*S*, **G**) is a pair (V, F) such that:

- V is an S-sorted set, and
- F = $\left\{ f_{\mathrm{g}} | \mathrm{g} \in \cup\Gamma \right\}$ is a set of functions such that

$$\mathrm{g} \in \Gamma_{<<s_0,\ldots,s_{n-1}>,r>} \Rightarrow f_{\mathrm{g}} \in V_{s_0} \times \ldots \times V_{s_{n-1}} \to V_r$$

$$\mathrm{g} \in \Gamma_{<s,r>} \Rightarrow f_{\mathrm{g}} \in V_s^{\,*} \to V_r$$

Set V is called the **carrier set** of the **S**-algebra, and set F is its **operator set**.

□

2.5 S-term algebra

Definition 2.10

A **S-term algebra** with respect to a signature with list operators **S** = (*S*, **G**) is the **S**-algebra ($Term_\Sigma$, F) such that:

- F = $\left\{ f_{\mathrm{g}} | \mathrm{g} \in \cup\Gamma \right\}$ is a set of functions (the operator set) such that

$$\mathrm{g} \in \Gamma_{<<s_0,\ldots,s_{n-1}>,r>} \Rightarrow f_{\mathrm{g}} \in Term_{\Sigma_{s_0}} \times \ldots \times Term_{\Sigma_{s_{n-1}}} \to Term_{\Sigma_r} \text{ and}$$

$$\left(\forall i \in N : i < n : t_i \in Term_{\Sigma_{s_i}} \right) \Rightarrow f_{\mathrm{g}}(t_0, \ldots, t_{n-1}) = \mathrm{g}[t_0, \ldots, t_{n-1}]$$

$$\mathrm{g} \in \Gamma_{<s,r>} \Rightarrow f_{\mathrm{g}} \in Term_{\Sigma_s}^{\ *} \to Term_{\Sigma_r} \text{ and}$$

n **Î** :(**"**i N: **<** n:t **Î** **s**): f_g($t_0,\ldots,$ $_{-1}$ **>**) **=** g[$t_0,\ldots,t$ $_{-1}$ **>**]

□

For instance, this is the operator set F of the **S**-term algebra for our running example:

$$F = \{ f_{ifelse}, f_{while}, f_{statlist}, f_{assignment}, f_a, f_b, \ldots, f_z, f_{True}, f_{False}, f_{value} \}$$

$f_{assignment}$ **Î** $Term_{\mathbf{s}_{Identifier}}$ **´** $Term_{\mathbf{s}_{Expr}}$ **®** $Term_{\mathbf{s}_{Stat}}$, $f_{assignment} : d, s \mapsto assignment[d, s]$

f_{ifelse} **Î** $Term_{\mathbf{s}_{Expr}}$ **´** $Term_{\mathbf{s}_{Stat}}$ **´** $Term_{\mathbf{s}_{Stat}}$ **®** $Term_{\mathbf{s}_{Stat}}$, $f_{ifelse} : g, st, sf \mapsto ifelse[g, st, sf]$

f_{while} **Î** $Term_{\mathbf{s}_{Expr}}$ **´** $Term_{\mathbf{s}_{Stat}}$ **®** $Term_{\mathbf{s}_{Stat}}$, $f_{while} : g, st \mapsto while[g, st]$

$f_{statlist}$ **Î** $Term_{\mathbf{s}_{Stat}}^{\ *}$ **®** $Term_{\mathbf{s}_{Stat}}$, $f_{statlist} : sl \mapsto statlist[sl]$

f_{value} **Î** $Term_{\mathbf{s}_{Identifier}}$ **®** $Term_{\mathbf{s}_{Expr}}$, $f_{value} : id \mapsto value[id]$

$f_{a\ldots z}$ **Î®** $Term_{\mathbf{s}_{Identifier}}$, $f_{a\ldots z} : \mapsto a\ldots z$

f_{False} **Î®** $Term_{\mathbf{s}_{Expr}}$, $f_{False} : \mapsto False$

f_{True} **Î®** $Term_{\mathbf{s}_{Expr}}$, $f_{True} : \mapsto True$

2.6 S-homomorphism

Definition 2.11

A **S-homomorphism** from **S**-algebra (V, F) to **S**-algebra (W, G) is an S-indexed set of functions *h* such that:

- For all $s \in S$ we have $h_s \in V_s \rightarrow W_s$ and
- For all aggregate operators $g \in \Gamma_{<<s_0,\ldots,s_{n-1}>,r>}$, $f_g \in F$, $g_g \in G$ and $\forall i \in N : i < n : e_i \in V_{s_i}$

  $$h_r(f_g(e_0,\ldots,e_{n-1})) = g_g(h_{s_0}(e_0),\ldots,h_{s_{n-1}}(e_{n-1}))$$
- For all list operators $g \in \Gamma_{<s,r>}$, $f_g \in F$, $g_g \in G$ and $e \in V_s^{\ n}$

  $$h_r\left(f_g(e)\right) = g_g\left(\left\langle h_{s_0}(e(0)),\ldots,h_{s_{n-1}}(e(n-1))\right\rangle\right)$$

□

In case there is only one sort, a **S**-homomorphism is a singleton set and we speak of the homomorphic function.

Definition 2.12

A **S**-algebra is **initial** if there is a unique **S**-homomorphism from it to all other **S**-algebras.

□

S-term algebras are initial.

2.7 Trees

Trees are very suitable to represent terms that are derived using a signature **S** (*S*, **G**). We will discuss a way to represent a term tree by a function having a 'tree domain' as a domain and **G** as range. Adapted and simplified from [2], Section 1.3.

2.7.1 Tree domains

Definition 2.13

A **tree node address** $a \in N^*$ of a node *n* is the sequence of steps to take, starting from the root node, to arrive at *n*. The address of the root node is **e**. Each step is represented by a nonnegative integer that indicates which of the current node's descendants to select. 0 indicates the leftmost descendant and *n*, where *n*>0, indicates the descendant immediately to the right of the descendant indicated by (*n*-1).

□

Definition 2.14

A **tree domain** D is a set of tree node addresses such that:

- ("*d* Î *D*:*d* ¹ e:*rd*(*d*,1) Î *D*)
- $(\forall d \in D, j \in N : d \oplus \langle j \rangle \in D : (\forall i \in N : i < j : d \oplus \langle i \rangle \in D))$

□

The height of a nonempty tree domain D is: $\left(\uparrow d : d \in D : |d|\right)$.

The following figure shows a tree with a node address for each node. The set of these node addresses {**e**, <0>, <0, 0>, <1>, <1, 0>, <1, 0, 0>, <1, 0, 0, 0>, <1, 0, 1>, <1, 0, 1, 0>, <1, 0, 1, 1>, <1, 0, 1, 2>, <1, 1>, <1, 1, 0>, <1, 1, 1>} is indeed a valid tree domain.

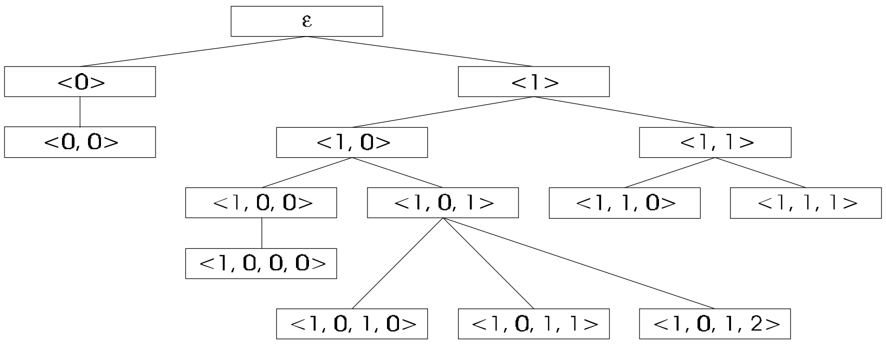


Figure 2.1 A tree domain.

2.7.2 Term trees

Given de above definitions, we can conveniently represent a term tree based on a signature **S** = (*S*, **G**) as a function from tree domain to operator set **ÈG**.

Definition 2.15

Given a signature **S** = (*S*, **G**) and a tree domain D, a term tree $f \in D \to \cup\Gamma$ and a node address *a*.

If $a \neq \mathrm{e}$, the node indicated by node address $rd(a,1)$ is called the **parent** of the node indicated by *a*. The nodes indicated by the node addresses $\{a \oplus < i > | a \oplus < i > \in D\}$ are called the **children** of the node indicated by *a*.

□

Definition 2.16

The S-sorted set $Tree_\Sigma$ of **syntax trees** or **term trees** based on a signature with list operators **S** = (S, **G**) is the smallest S-sorted set such that:

$\forall n \in N :: \forall t_0 \ldots t_{n-1} \in \cup Tree_\Sigma ::$

$$(\forall \mathrm{g}, a \in S^n, r \in S : \mathrm{g} \in \Gamma_{<a,r>} : (\forall i : i \in N \wedge i < n : t_i \in Tree_{\Sigma_{a(i)}}) \Rightarrow$$
$$\{(\mathrm{e},\mathrm{g})\} \cup \{(\langle i \rangle \oplus v, w) | i \in N \wedge i < n \wedge (v,w) \in t_i\} \in Tree_{\Sigma_r})$$

$$(\forall \mathrm{g}, a \in S, r \in S : \mathrm{g} \in \Gamma_{<a,r>} : (\forall i : i \in N \wedge i < n : t_i \in Tree_{\Sigma_a}) \Rightarrow$$
$$\{(\mathrm{e},\mathrm{g})\} \cup \{(\langle i \rangle \oplus v, w) | i \in N \wedge i < n \wedge (v,w) \in t_i\} \in Tree_{\Sigma_r})$$

□

There is a one-to-one correspondence between $Term_\Sigma$ and $Tree_\Sigma$. Please refer to [2], page 17, for a proof. The above definition expresses that, in a term tree, for each node, its number of children must be equal to the arity of the corresponding operator if it is an aggregate operator. Furthermore, the sorts of a node's children must equal the operator argument sorts.

For example, the only term tree (element of $Tree_{\Sigma_{Stat}}$) that corresponds to the example term given in section 2.4 (*ifelse[value[a],assignment[b,False],assignment[c,True]]*) is:

{ (**e**, *ifelse*),
(<0>, *value*),
(<0, 0>, *a*),
(<1>, *assignment*),
(<1, 0>, *b*),
(<1, 1>, *False*),
(<2>, *assignment*),
(<2, 0>, *c*),
(<2, 1>, *True*)
}

Or in graphical form:

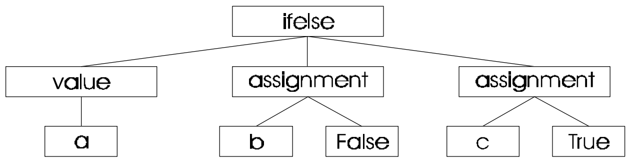


Figure 2.2 Term tree

2.8 Structure Editing

The purpose of a structure editor is to let the user construct a term that conforms to a predefined signature (*S*, G). The term *t* in the editor is of a fixed predefined sort TERMSORT **Î** *S*, so *t* **Î** $Tree_{\mathbf{S}_{TERMSORT}}$.

While a term is under construction in a structure editor, it will have missing subterms. To be able to represent a term with one or more missing subtrees, for each sort s**Î**S exactly one *hole operator* **D**$_s$ (a constant) is included in the signature's operator set. **D** is the S-indexed set of hole operators (also referred to as *holes*) and, obviously, |**D**| = |S|and **D Ì ÈG**. For every missing subtree, a hole operator of the same sort will appear in the term tree.

For example, a structure editor would use the following extended operator set for our running example signature:

ÈG= {*StatHole, ExprHole, IdentifierHole, ifelse, while, statlist, assignment, a, b, ..., z, True, False, value*}

D = {**D**$_{Stat}$, **D**$_{Expr}$, **D**$_{Indentifier}$},

D$_{Stat}$:= *StatHole*
D$_{Expr}$:= *ExprHole*
D$_{Identifier}$:= *IdentifierHole*

Definition 2.17

A term tree t**Î** *Tree*$_\Sigma$ is called **complete** if and only if $rng(t) \cap \Delta = \varnothing$.

□

We will now discuss the basic editing operations on a term tree.

2.8.1 Subtree replacement

Definition 2.18

Assume *I* to be any nonempty set, T to be any type and $f \in I^* \to T$.

The **translation** of a function $I^* \to T$ by a sequence I^* is the result of the -infix denoted-function $\tilde{\oplus}: I^* \times (I^* \to T) \to (I^* \to T)$.

Given $f \in I^* \to T$ and $s \in I^*$, then

$$s \tilde{\oplus} f := \{(s \oplus o, q) \mid (o, q) \in f\}$$

□

Definition 2.19

Assume **S** to be a signature (*S*, G).

The **replacement** of a subtree at address $a \in N^*$ of a term tree $t \in Tree_\Sigma$ by a term tree $u \in Tree_\Sigma$ is the result of the function $replace: Tree_\Sigma \times N^* \times Tree_\Sigma \to Tree_\Sigma$

Given $a \in D(t) \wedge (\exists s \in S :: t(a) \in \Gamma_s \wedge u(\text{e}) \in \Gamma_s)$

$$replace(t_0, a, t_1) := \{(o, p) \mid (o, p) \in t_0 \wedge \neg(a \leq_p o)\} \cup (a \tilde{\oplus} t_1)$$

□

Example:

replace(*ifelse[value[a],assignment[b,False],assignment[c,True]]*, <2, 1>, *value[d]*)=

ifelse[value[a],assignment[b,False],assignment[c,value[d]]]

Deleting a subtree is accomplished by replacing it by a hole of the same sort.

For example:

replace(*ifelse[value[a],assignment[b,False],assignment[c,True]]*, <2, 1>, *ExprHole*)=

ifelse[value[a],assignment[b,False],assignment[c,ExprHole]]

2.8.2 Subtree extraction

Definition 2.20

Assume **S** to be a signature (*S*, G).

The extraction of a subtree at address $a \in N^*$ from a term tree $t \in Tree_\Sigma$ is the result of the function $subtree: Tree_\Sigma \times N^* \to Tree_\Sigma$.

Given $a \in D(t)$

$$subtree(t, a) := \{(o, p) \mid (a \oplus o, p) \in t\}$$

□

Example:

subtree(*ifelse[value[a],assignment[b,False],assignment[c,True]]*, <2>)=

assignment[c,True]

2.8.3 Refining

One of the most important editing operations is refining. Its purpose is to replace a hole by an operator of the same sort. If and only if the new operator is an aggregate operator with arguments, appropriate holes must be created at the same time to ensure that the tree remains a valid term tree.

Definition 2.21

Assume **S** to be a signature (*S*, *G*).

The **refining** of a subtree at address $a \in N^*$ of a term tree $t \in Tree_\Sigma$ with an operator **g Î ÈG** is the result of the function $refine\colon Tree_\Sigma \times N^* \times \Gamma \to Tree_\Sigma$.

Given $a \in D(t) \wedge s \in S^* \wedge \left(\exists r\colon r \in S\colon t(a) \in \Delta_r \wedge \mathrm{g} \in \Gamma_{<s,r>}\right)$,

$$refine(t,a,\mathrm{g}) := replace\left(t, a, \{(\mathrm{e},\mathrm{g})\} \cup \left\{(<i>, \Delta_{s(i)}) \middle| i \in N \wedge i < |s|\right\}\right),$$

And given $a \in D(t) \wedge \left(\exists r, s\colon r \in S, s \in S\colon t(a) \in \Delta_r \wedge \mathrm{g} \in \Gamma_{<s,r>}\right)$,

$$refine(t,a,\mathrm{g}) := replace\left(t, a, \{(\mathrm{e},\mathrm{g})\}\right).$$

□

Example:

refine(*ifelse[value[a],assignment[b,False],StatHole]*, <2>, *while*)=

ifelse[value[a],assignment[b,False],while[ExprHole,StatHole]]

2.8.4 List insertion

A list operator has $n \in N$ children, numbered from 0 through *n*-1, all of the same sort. The structure editor will have to let the user insert a new child between existing children or at the beginning or end of the list.

Definition 2.22

Assume **S** to be a signature (*S*, **G**).

The following function inserts a subtree $u \in Tree_\Sigma$ so that it becomes the child $n \in N$ of the list operator at address $a \in N^*$ of a term tree $t \in Tree_\Sigma$:

$insertchild\colon Tree_\Sigma \times N^* \times N \times Tree_\Sigma \to Tree_\Sigma$. Given

$$a \ \text{Î}\ D(t)\ \text{Ù}\ c = \left\{\left(i, subtree(t, a\,\text{Å} <i>)\right) \middle| a\,\text{Å} <i> \text{Î}\ D(t)\right\} \text{Ù}\ n\ \text{£}\ |c|\ \text{Ù}$$
$$IsListOp\left(t(a)\right) \text{Ù} \left(\text{\$}r\ \text{Î}\ S\colon t(a)\ \text{Î}\ \text{G}_r\ \text{Ù}\ u\ \text{Î}\ Tree_{\text{S}_s}\right)$$

then $insertchild(t,a,n,u) := replace\left(t, a, \{\mathrm{e}, t(a)\} \cup \left\{<i> \tilde{\oplus} insert(c,n,u)(i) \middle| i \in N \wedge i \lhd |c|+1\right\}\right)$

□

Example:

insertchild(*StatList[assignment[b,False],assignment[c,True],StatList]*, **e**, 1, *StatHole*)=

StatList[assignment[b,False],StatHole,assignment[c,True],StatList]

2.8.5 List deletion

The reverse operation should be offered too: deleting the *n*'th child of a list operator at a given address in the term tree.

Definition 2.23

Assume **Σ** to be a signature (*S*, Γ).

The following function deletes a subtree, child $n \in N$ of the list operator at address $a \in N^*$ of a term tree $t \in Tree_\Sigma$: $delete\colon Tree_\Sigma \times N^* \times N \to Tree_\Sigma$.

Given $a \in D(t) \wedge c = \{(i, subtree(t, a \oplus <i>)) | a \oplus <i> \in D(t)\} \wedge n < |c| \wedge IsListOp(t(a))$,

$$deletechild(t,a,n) := replace\left(t, a, \{\varepsilon, t(a)\} \cup \{<i> \tilde{\oplus} delete(c,n)(i) | i \in N \wedge i < |c| - 1\}\right)$$

□

Example:

deletechild(*StatList[assignment[b,False],StatHole,assignment[c,True],StatList]*, **ε**, 0)=

StatList[StatHole,assignment[c,True],StatList]

2.9 View terms over a signature

The aim is to develop a structure editor with multiple views of several different types on the term tree. For each view a data structure is needed of exactly the same shape as the term tree, with the requirement that each node carries a value (an element of a set specific to the view type). Furthermore, a view needs a set of functions where each function is associated with an operator; it computes the associated value of a node from the values associated with each of the operator's children. This data structure will be referred to as a *view tree* or *view term*.

Analogous to 2.3, the following definition concerns the set of view terms that can be derived using a signature.

Definition 2.24

Given a signature **Σ** = (*S*, **Γ**),

a set I (specific to a view type, containing the values that can be associated with each operator in the view term)

a set $D = \{d_\gamma | \gamma \in \cup\Gamma\}$,

- for each operator $\gamma \in \Gamma_{<s,r>}, s \in S^*, r \in S$, $d_\gamma \in I^{|s|} \to I$.
- for each operator $\eta \in \Gamma_{<s,r>}, s \in S, r \in S$, $d_\eta \in I^* \to I$

The S-sorted set $ViewTerm_{\Sigma,I,D}$ is the smallest *S*-sorted set such that

$$\forall n \in N ::$$

$$(\forall \gamma, a \in S^n, r \in S : \gamma \in \Gamma_{<a,r>} : (\forall l : l \in N \wedge l < n : t_l \in ViewTerm_{\Sigma,I,D_{a(l)}}) \Rightarrow$$

$$\left(\gamma\left[t_0, \ldots, t_{n-1}\right], d_\gamma(\pi_2(t_0), \ldots, \pi_2(t_{n-1}))\right) \in ViewTerm_{\Sigma,I,D_r})$$

$$\wedge$$

$$(\forall \gamma, s \in S, r \in S: \gamma \in \Gamma_{\langle s,r \rangle}: (\forall l: l \in N \wedge l < n: t_l \in ViewTerm_{\Sigma,I,D_s}) \Rightarrow$$

$$\left(\gamma\left[\langle t_0, \ldots, t_{n-1} \rangle\right], d_\gamma(\langle \pi_2(t_0), \ldots, \pi_2(t_{n-1}) \rangle)\right) \in ViewTerm_{\Sigma,I,D_r})$$

The square brackets are used syntactically here. We adopt the convention that $\gamma[]$ is simply written as γ. The operator π is used for tuple projection. Given a tuple $t = (x_1, \ldots, x_n)$ we can denote tuple element x_i, $i \in N, 1 \leq i \leq n$ of t by $\pi_i(t)$.

□

2.10 Σ-view algebra

Analogous to the concept of **Σ**-term algebra, defined in Section 2.5, we will now define a **Σ**-view algebra, given a set I (specific to a view type, containing the values that can be associated with each operator in the view term) and a set of functions D.

Definition 2.25

A **Σ-view algebra** is the **Σ**-algebra $(ViewTerm_{\Sigma,I,D}, F(I,D))$ such that:

- I is a set .
- $D = \left\{d_\gamma \middle| \gamma \in \bigcup\Gamma\right\}$ is a set of functions,
  - for each operator $\gamma \in \Gamma_{<s,r>}, s \in S^*, r \in S$, $d_\gamma \in I^{|s|} \to I$.
  - for each operator $\eta \in \Gamma_{<s,r>}, s \in S, r \in S$, $d_\eta \in I^* \to I$
- $F(I,D)$ = $\left\{f_\gamma | \gamma \in \bigcup\Gamma\right\}$ is a set of functions such that

$$\gamma \in \Gamma_{\langle\langle s_0,\ldots,s_{n-1}\rangle,r\rangle} \wedge (\forall l \in N: l < n: t_l \in ViewTerm_{\Sigma_{s_l}}) \Rightarrow$$

$$f_\gamma(t_0, \ldots, t_{n-1}) = \left(\gamma[t_0, \ldots, t_{n-1}], d_\gamma(\pi_2(t_0), \ldots, \pi_2(t_{n-1}))\right)$$

$$\gamma \in \Gamma_{\langle s,r\rangle} \Rightarrow \forall n \in N: (\forall i \in N: i < n: t_i \in ViewTerm_{\Sigma_s}):$$

$$f_\gamma(\langle t_0, \ldots, t_{n-1} \rangle) = \left(\gamma[\langle t_0, \ldots, t_{n-1} \rangle], d_\gamma(\pi_2(t_0), \ldots, \pi_2(t_{n-1}))\right)$$

□

2.11 Homomorphism from Σ-term to Σ-view algebra

Assume we have a **Σ**-term algebra ($Term_\Sigma$, F) and a **Σ**-view algebra ($ViewTerm_{\Sigma,I,D}$, $G(I,D)$). The homomorphism from **Σ**-term algebra ($Term_\Sigma$, F) to a **Σ**-view algebra ($ViewTerm_{\Sigma,I,D}$, $G(I,D)$) is an S-indexed set of functions $h(I,D)$ such that:

For all $s \in S$ we have $h(I,D)_s \in Term_{\Sigma_s} \to ViewTerm_{\Sigma,I,D_s}$

$$\forall s \in S, a \in S^*, \gamma \in \Gamma_{\langle a,s\rangle}: \left(\forall j: 0 \leq j < |a|: t_j \in Term_{\Sigma_{a(j)}}\right):$$

$$h(I,D)_s(\gamma[t_0, \ldots, t_{|a|-1}]) = g_\gamma\left(h(I,D)_{a(0)}(t_0), \ldots, h(I,D)_{a(|a|-1)}(t_{|a|-1})\right)$$

and

$$\forall s \in S, a \in S, \gamma \in \Gamma_{\langle a,s\rangle}, n \in N: \left(\forall j: 0 \leq j < n: t_j \in Term_{\Sigma_a}\right):$$

$$h(I,D)_s(\gamma[\langle t_0, \ldots, t_{n-1} \rangle]) = g_\gamma\left(\langle h(I,D)_a(t_0), \ldots, h(I,D)_a(t_{n-1}) \rangle\right)$$

By definition, this conforms to the conditions in the homomorphism definition (Definition 2.11).

3 Implementation environment

Some of the most important requirements for the structure editor to be developed are the ability to simultaneously have multiple views on the term being edited and the possibility to have different types of views, each of which can consist of a mixture of graphics and text. To meet these requirements, the structure editor needs a graphical user interface (GUI). Since the most widespread GUI today is Microsoft Windows on IBM-compatible PC's (referred to as Windows from now on), this will be the implementation platform of choice for the structure editor.

Another decision to be made is choosing the programming language. The mathematical concepts discussed in chapter 2 lend themselves well to implementation in an object-oriented language. This work builds on an early structure editor implementation in Borland's object-oriented PASCAL. An well-known alternative to this language is C++, an object-oriented successor of C, which has some powerful features that object-oriented PASCAL lacks. It was decided to go with C++ and set as one of the goals to find out how its features can be used to model and implement **S**-algebras in an efficient, concise and elegant way.

The last implementation environment decision is choosing a C++ *application framework*. It is possible to create Windows applications without any layer between C++ and Windows by directly calling the Windows Application Programmer's Interface (API), which is a collection of functions and procedures that allow an application to use Windows. However, the API was designed with structured programming in C and PASCAL in mind, before object-oriented programming became popular. The C++ application framework provides a layer of abstraction over the Windows API. It consists of a class library that encapsulates the Windows API and makes it possible to write completely object-oriented Windows applications. The two best-known C++ application frameworks for Windows today are Borland's *ObjectWindows Library (OWL)* and *Microsoft Foundation Classes (MFC)*. OWL has a strong object-oriented design, uses C++'s features well and is considered technically superior to MFC. This, and the fact that TUE has a Borland campus licence made OWL the obvious choice as the application framework for the structure editor.

This chapter will describe the features of C++ and ObjectWindows that are relevant to the implementation of the structure editor.

3.1 C++ features

Except for minor details, C++ is a superset of the C programming language. It is designed to support *data abstraction* and *object-oriented programming*, while retaining C's efficiency. The language is described by its author in [5].

Data abstraction (sometimes called *data hiding*) allows a programmer to define a type together with a full set of operations. As opposed to record types in procedural languages, the actual data in an instance of an abstract data type is hidden to the programmer and can only be manipulated by using the publicly available set of operations for the type.

Object-oriented programming lets a programmer derive new types from existing types by using *inheritance* and it offers a mechanism that allows calls of operations on a type to depend on the actual type of the object (in cases where the actual type is unknown at compile time).

3.1.1 Classes

A class is a user-defined type. It represents a unique set of objects or *data members* (just like a record type in procedural languages) and the operations (called *member functions*, or *methods*) to create, manipulate, convert and destroy these objects.

For example, here is a stack class to illustrate the data abstraction of C++ classes:

```
class TCharStack {
private:               // label is redundant: private by default
   int     size;
   char   *top;
   char   *s;
public:
                 TCharStack(int sz);
                ~TCharStack();
   inline void   push(char c);
   inline char   pop();
};
```

Throughout this project, the convention is to start the names of classes that are defined with ‘T’.

The data members are `size`, `top` and `s`. `TCharStack`, `~TCharStack`, `push` and `pop` are member functions. The `public:` label indicates the start of the publicly available members (the interface) of the class. The member function with the same name as the class is called *constructor*; its purpose is to initialize the object whenever an instance of the class is created. The inverse of this is the *destructor*, which ensures proper cleanup of objects of this class. It is indicated by ~ (complement) followed by the class name. In this case, `~TCharStack()` is the destructor. Constructors and destructors have no return type specification.

The overhead of a function call will be a significant factor in the total execution time of a member function when the member function itself is very simple. To avoid this overhead, a member function can be declared `inline`. This means that each time the member function is invoked, the compiler inserts the actual code into the body of the calling function instead of a function call.

The definitions of the member functions declared above will complete the definition of TCharStack:

```
TCharStack::TCharStack(int sz)
{
   size = sz;
   top = s = new char[size];  // allocate array of size characters from free store
}

TCharStack::~TCharStack()
{
   delete[] s;
}

inline void      TCharStack::push(char c)
{
   *top++ = c;
}

inline char      TCharStack::pop();
{
   return *--top;
}
```

The `new` and `delete` operators allocate and free objects from the free store, respectively.

It will now be shown how an object of class TCharStack can be instantiated and used:

```
void f()
{
   char       b, a;
   char      *p;
   TCharStack      s(100);      // The TCharStack constructor is called with 100 as its argument

   s.push(‘A’);
   s.pusb(‘B’);
   b = s.pop();
   a = s.pop();    // a == ‘A’ and b==‘B’

   p = s.top;      // illegal, top is a private data member of TCharStack.
   s.s = 0;        // s also.
   // s will go out of scope and the destructor ~TCharStack will be called
}
```

3.1.2 Derived classes

In C++, it is possible to create a class by *deriving* from a number of existing classes and adding facilities to it. The classes being derived from are called *base classes***.** A situation where a class has one direct base class is called *single inheritance*. Having multiple direct base classes is called *multiple inheritance*. A derived class is sometimes called a *subclass* and a base class a *superclass*.

Consider the TCharStack class. It offers facilities to create a stack of characters and do push and pop operations for one character at a time. A logical extension of this would be the TStringStack class. Since a string is represented as a sequence of characters terminated by the 0 character, we can reuse the TCharStack by deriving TStringStack from it. This leads to the following declaration:

```
class TStringStack: public TCharStack {
public:
                  TStringStack(int sz);
   void           push(const char *string);
   void           pop(char *string);
};

TStringStack::TStringStack(int sz) : TCharStack(sz)
{
}

void        TStringStack::push(const char *string)
{
   char   *end = string+strlen(string);
   do {
      push(*end--);
   } while(string<=end);
}

void        TStringStack::pop(char *string)
{
   while(*string++ = pop())
      ;
}
```

The `public` specifier that precedes the name of the base class indicates *public inheritance*; all public member functions of the base class remain accessible to users of the derived class. The constructor simply passes on its argument to the TCharStack constructor. No destructor exists because the TCharStack destructor already takes care of everything. In this simple case, no additional cleanup is needed.

The `push` and `pop` member functions are now overloaded. Depending on the argument given, either the TCharStack or the TStringStack version is called. The TStringStack `push` member function pushes the string characters from right to left (using the TCharStack version of `push`), starting with the string terminating 0. The `pop` member function rebuilds the string by popping characters off the stack until it has popped the terminating 0.

I will now show how an object of class TStringStack can be instantiated and used:

```
void f()
{
   char       a;
   char       str[10];           // should fit longest string ever popped off string stack.

   TStringStack      s(100);      // The TStringStack constructor passes 100 to TCharStack
                                 // constructor.

   s.push(“aap”);
   s.pusb(“noot”);
   s.pop(str);                  // str = “noot”
   a = s.pop();                  // use TCharStack version of pop(), a = ‘a’
   s.push(‘A’);                  // use TCharStack version of push()
   s.pop(str);                  // str = “Aap”
// s will go out of scope and the destructor ~TCharStack will be called
}
```

Since the class TStringStack includes all data and member functions of TCharStack, a TStringStack * (pointer to a TStringStack object) can always be implictly converted to a TCharStack * wherever needed. Obviously, the opposite is not true.

A derived class can itself be a base class so that there can be multiple levels of inheritance. A set of related classes is often called a *class hierarchy*. In C++, this hierarchy has the form of a directed acyclic graph, where the edges go from each class to all of its direct base classes.

3.1.3 Polymorphism

If a base class `b` contains a member function `func` declared with `virtual` specifier and a class `d` derived from it also contains a function `func` of the same type, then the call of `func` for an object of class `d` invokes `d::func`, even if the call is through a pointer or reference to the base class `b`. In this case, it is not known at compile time what the actual type of the object pointed to by a base class pointer is. The actual method code to call is determined at run time by *late* or *dynamic* binding.

Example:

```
class TShape {
public:
   virtual char    *Name() = 0;
};

class TSquare: public TShape {
public:
   char            *Name() { return “Square”; }
};

class TCircle: public TShape {
public:
   char            *Name() { return “Circle”; }
};

class TLine: public TShape {
public:
   char            *Name() { return “Line”; }
};

void f()
{
   char      *p;
   TShape   *shapes[5];

   shapes[0] = new TCircle;
   shapes[1] = new TCircle;
   shapes[2] = new TSquare;
   shapes[3] = new TLine;
   shapes[4] = new TCircle;

   p = shapes[0]->Name();         // p=“Circle”;
   p = shapes[2]->Name();         // p=“Square”;
   p = shapes[3]->Name();         // p=“Line”;

   for(int i = 0; i<5; i++)
      delete shapes[i];
}
```

The class TShape does not describe a shape; it expresses the commonality between the actual shape classes TSquare, TCircle and TLine. To prevent a meaningless instantiation of TShape, the virtual function Name() is declared as *pure virtual* by including ‘=0’ in the declaration. A class which has at least one pure virtual function is called an *abstract class* and cannot be instantiated. In this case, we may refer to TShape as an *abstract base class* of the actual shape classes. The following declarations cause compile-time errors:

```
TShape    s;
TShape   *p = new TShape;
```

3.1.4 Templates

A C++ *template* defines a family of classes or functions. By allowing the programmer to use parameterized types, the concept of class templates allows container classes such as stacks, lists and associative arrays to be defined and implemented in a simple way, without loss of static type checking or run-time efficiency. Similarly, templates allow generic functions to be defined once for a family of types.

Class templates

The first C++ example given was the class TCharStack, a stack of `char` objects. In a similar way, we could define a stack of `int`s, `short`s, `double`s, `long`s, `complex’`s, etcetera. This leads to many similar class declarations, differing only in the type of the objects that can be stored on the stack.

Using templates, it is easy to define a stack of elements of any type:

```
template<class T>
class TStack {
   int    size;
   T      *top;
   T      *s;
public:
                  TStack(int sz);
                 ~TStack();
   inline void    push(T c);
   inline T       pop();
};

template<class T>
TStack<T>::TStack(int sz)
{
   size = sz;
   top = s = new T[size];  // allocate array of size objects of type T from free store
}

template<class T>
TStack<T>::~TStack()
{
   delete[] s;
}

template<class T>
inline void      TChar<T>::push(T c)
{
   *top++ = c;
}

template<class T>
inline T         TChar::pop()
{
   return *--top;
}
```

It will now be shown how stacks of elements of various types can be made using the TStack class template:

```
void f()
{
   TStack<char>     s(100);
   TStack<double>   d(20);

   s.push(‘A’);
   d.push(3.1415927);
// d and s will go out of scope and their destructors will be called
}
```

Function templates

Similar to class templates, function templates can be used to define generic functions once and use them on a wide variety of types. A famous example is the function `max`, defined in `stdlib.h`, that works for every type which has a comparison operator >:

```
template<class T>
inline const T&   max(const T& t1, const T& t2)
{
   if(t1>t2)
      return t1;
   else
      return t2;
}
```

A template is only used to form an actual function declaration if one doesn’t exist already for the type(s) used as parameters to the template. This allows us to treat special cases in a sensible way. For instance, calling the template version of `max` with string arguments leads to a meaningless comparison between two character pointers. To avoid this, a nontemplate function can be used to override the generation of a template function for the type `char *`.

```
inline char   *max(char *t1, char *t2)
{
   if(strcmp(t1, t2)>0)
      return t1;
   else
      return t2;
}
```

3.2 ObjectWindows Library

ObjectWindows Library (*OWL*) is the Borland C++ application framework for Windows. Version 2.0 was used for the structure editor development.

The OWL class hierachy covers most of the Windows API and provides high-level object-oriented encapsulations for GDI graphics, Windows and MDI, as well as complete support for a Document/View architecture. It is beyond the scope of this document to give explanations for all OWL classes, but, to give a quick overview, the following 2 diagrams depict the entire OWL class hierarchy. The classes are grouped according to functional categories, and all related classes are in one shaded unit.

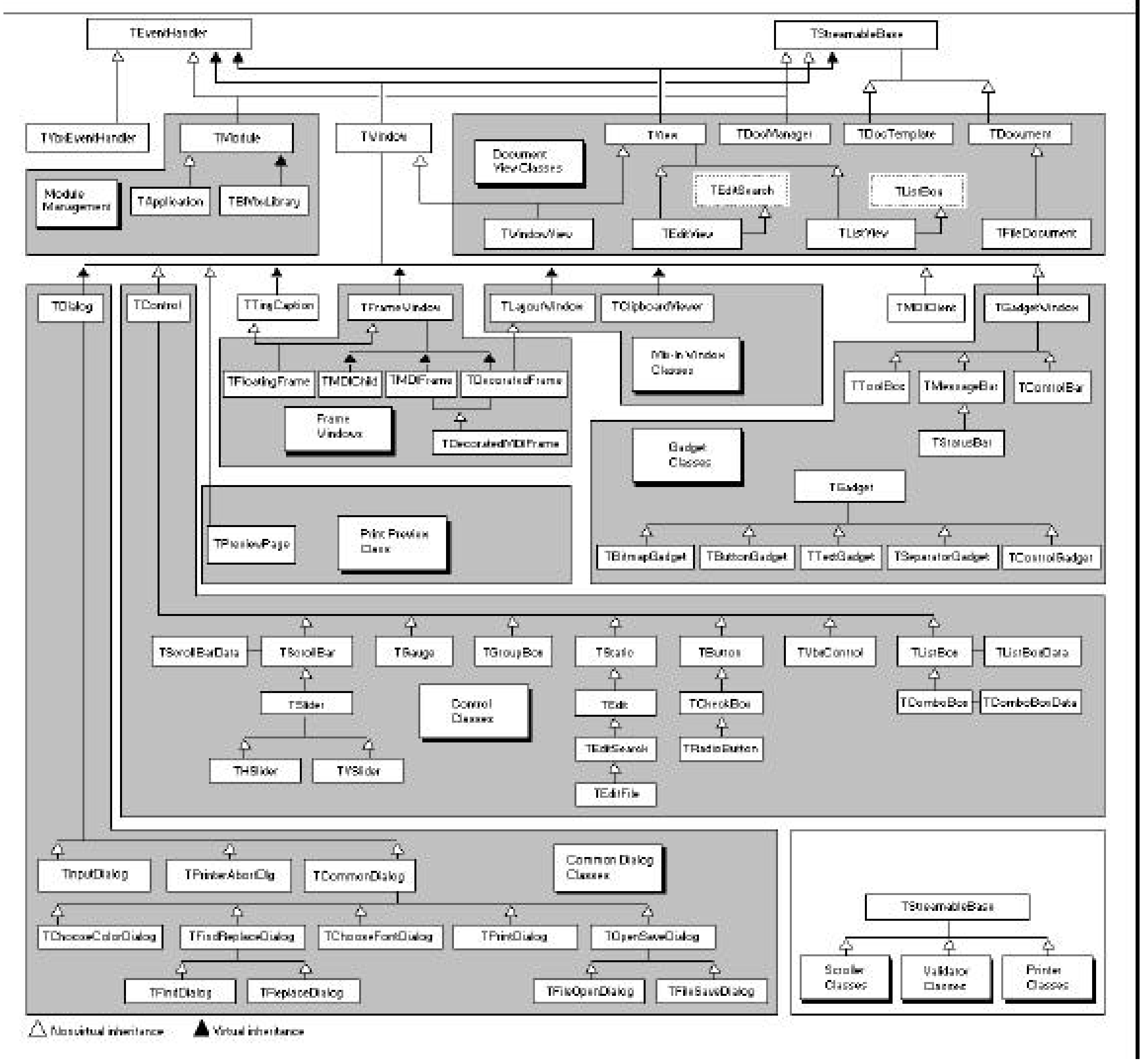


Figure 3.1 ObjectWindows class hierarchy diagram 1

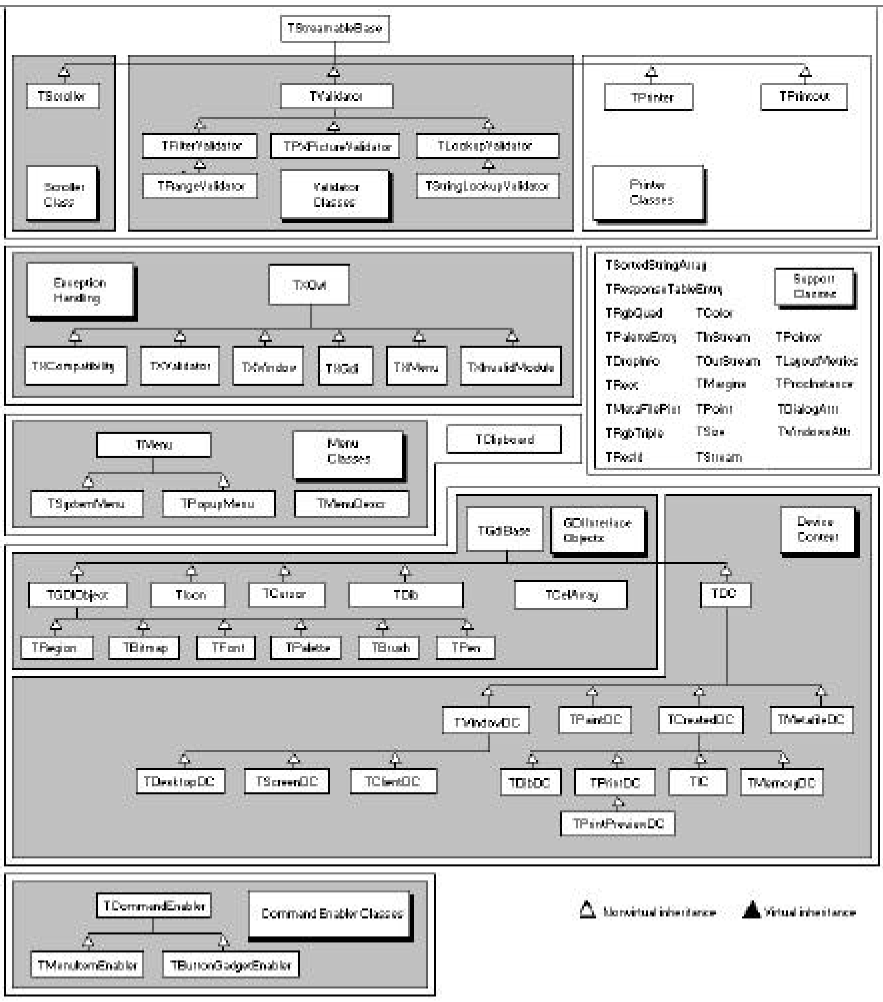


Figure 3.2 ObjectWindows class hierarchy diagram 2

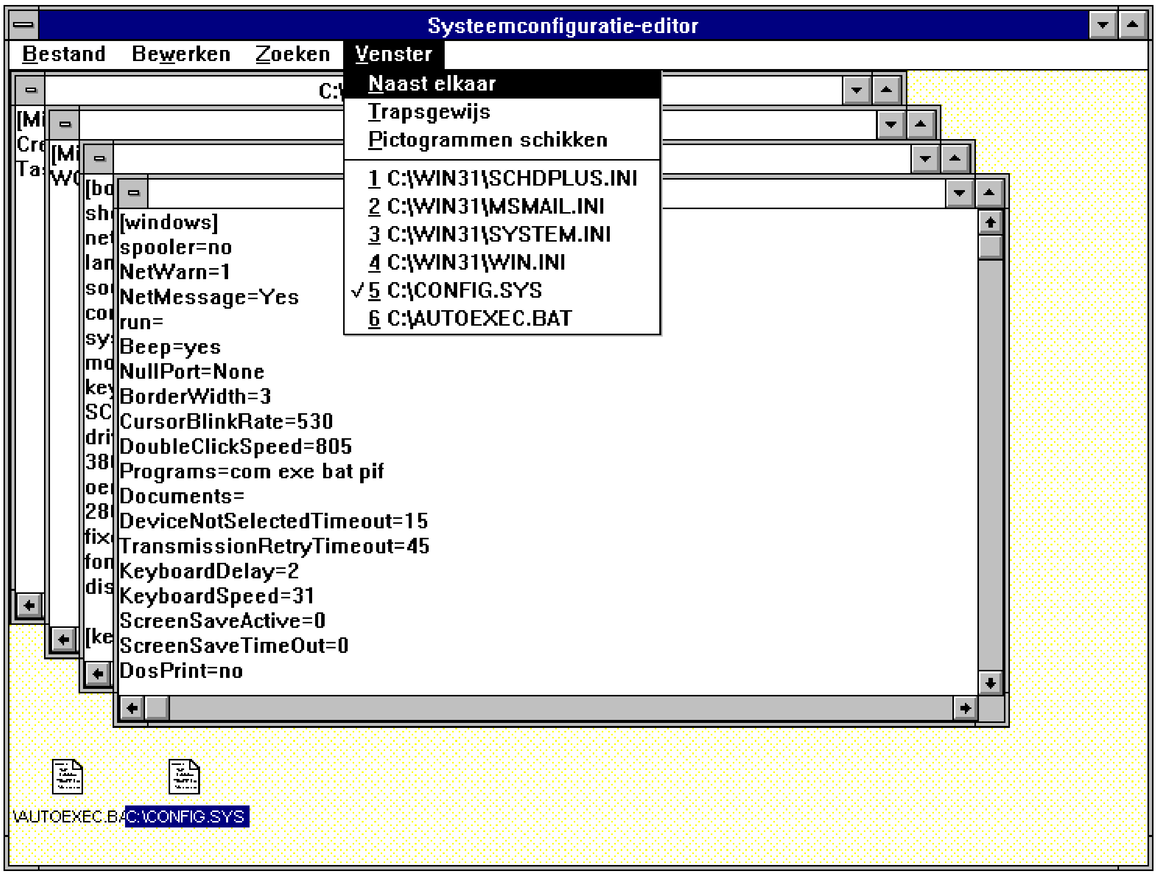
Only those major Windows and OWL features are discussed that will be essential in understanding the structure editor implementation. For a full ObjectWindows Library reference, please take a look at [6].

3.2.1 Multiple Document Interface

MS-Windows supports the *Multiple Document Interface (MDI)* user interface standard for presenting and manipulating multiple documents or views on documents within a single application. The structure editor is an MDI application; it has one main window (the MDI frame window) within which the user can work with several documents and views on documents. Each of these appears in its own separate child window (MDI child window) within the main application window.

Each MDI child window has a frame, system menu, maximize and minimize buttons, an icon and a title bar. The title bar shows the name of the document the MDI child window belongs to. The user can manipulate MDI child windows just as if they were normal, independent windows. However, MDI child windows cannot move outside the main application window.

The following illustration shows the SysEdit utility that comes with Windows 3.1; it is a good example of an MDI application.



```
[windows]
spooler=no
NetWarn=1
NetMessage=Yes
run=
Beep=yes
NullPort=None
BorderWidth=3
CursorBlinkRate=530
DoubleClickSpeed=805
Programs=com exe bat pif
Documents=
DeviceNotSelectedTimeout=15
TransmissionRetryTimeout=45
KeyboardDelay=2
KeyboardSpeed=31
ScreenSaveActive=0
ScreenSaveTimeOut=0
DosPrint=no
```

Figure 3.3 The SysEdit utility

As the picture shows, MDI child windows can be iconified. The ‘Window’ menu lists all the MDI child windows. Each window entry shows a document name and, when there are multiple views on that document, a colon followed by the view number (starting with 1).

3.2.2 Windows Clipboard

The clipboard is an important data-exchange feature of Windows; it is a common area to store data handles through which applications can exchange formatted data. The clipboard holds any number of different data formats and corresponding data handles. Most Windows applications, including the structure editor, offer the following three menu items for interaction with the clipboard:

Copy:	copies selected data from a document to the clipboard (replacing the clipboard’s previous contents)
Cut:	copies selected data onto the clipboard (using *Copy*) and then deletes the data from the document
Paste:	if there is suitable data on the clipboard, insert it into the document, leaving the clipboard unchanged

In OWL, the class TClipboard can be used to access the Windows clipboard. The following code fragments show how to place text on the clipboard and retrieve it later.

```
void     put_hello(window_handle)
HWND      window_handle;
{
   HGLOBAL     handle  = 0;
   char       *p = 0;
   TClipboard  cb(window_handle);            // TClipboard constructor needs a HWND

   handle  = ::GlobalAlloc(GHND, 6);       // Use windows function to get global memory.
   if(!handle)
      goto quit;
   p = (char *)::GlobalLock(handle);
   if(!p)
      goto quit;
   strcpy(p, “Hello”);
   if(!cb.EmptyClipboard())                 // Empty clipboard first
      goto quit;

   ::GlobalUnlock(handle);
   p = 0;
   cb.SetClipboardData(CF_TEXT, handle);    // Place data on clipboard
   handle = 0;                              // Do not free handle if placed successfully.
quit:
   if(p)
      ::GlobalUnlock(handle);
   if(handle)
      ::GlobalFree(handle);
}


void     get_string(window_handle, str)
HWND     window_handle;
{
   HGLOBAL     handle;
   char       *p = 0;
   TClipboard  cb(window_handle);            // TClipboard constructor needs a HWND

   handle  = cb.GetClipboardData(CF_TEXT);   // Any text on clipboard ?
   if(!handle)
      goto quit;
   p  = (char *)::GlobalLock(handle);
   if(!p)
      goto quit;
   strcpy(str, p);                           // copy the string
quit:
   if(p)
      ::GlobalUnlock(handle);
}
```

3.2.3 Message handling

An OWL application works *event-driven*, which means that the application is set up to respond to a wide variety of event types. Most event types are related to user actions, such as mouse clicks, keyboard input and menu item selections. When an event occurs, Windows notifies the application(s) to which it applies by sending them a message.

The Windows event handling poses special problems for C++ class hierarchies. Windows sends messages to a C callback function, where the message is decoded and processed in accordance with the message type. To map Windows messages to the correct C++ member function ObjectWindows uses entities known as *response tables*. These tables provide a connection between a Windows message and a C++ member function. To use response tables with a window, you must declare the table in the window's class declaration, and define the table in later the source file, outside the class declaration. Here is how a response table is declared in a sample window class:

```
class TMyWindow : public TFrameWindow {
public:
    // ...
     void       EvLButtonDown(UINT ModKeys, TPoint& point);
     void       CmAbout();

     DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

The definition of the response table is put in the source file, and connects Windows messages to member functions of class TMyWindow. The definition would look like this:

```
DEFINE_RESPONSE_TABLE1(TMyWindow, TFrameWindow)
   EV_WM_LBUTTONDOWN,
   EV_COMMAND(CM_ABOUT, CmAbout),
END_RESPONSE_TABLE;
```

ObjectWindows supports Windows messages like `WM_PAINT`, user commands sent through `WM_COMMAND` messages, notification messages, etc. Standard Windows messages are mapped automatically to ObjectWindows member functions, so you don't have to specify their name on the response table. For example, the `WM_LBUTTONDOWN` message uses the `EV_WM_LBUTTONDOWN` macro to connect the message with the member function `TMyWindow::EvLButtonDown()`. For user commands, such as menu selections, you must specify the member function name, using the `EV_COMMAND` macro. In the example, the `CM_ABOUT` menu selection would be handled by the function `TMyWindow::CmAbout()`.

For better understanding, the types of messages processed by the structure editor implementation will be listed here:

`EV_COMMAND(`*id, member func*`)`	handler for menu selections & accelerator keys
`EV_COMMAND_AND_ID(`*id, member func*`)`	handler for multiple commands using one func
`EV_WM_PAINT`	repaint (part of) a window
`EV_WM_CHAR`	ASCII keyboard characters
`EV_WM_LBUTTONDOWN`	left mouse button down
`EV_WM_RBUTTONDOWN`	right mouse button down
`EV_WM_LBUTTONDBLCLK`	left mouse button double click
`EV_WM_KEYDOWN`	key down (passed to app as virtual key code)
`EV_WM_KEYUP`	key up (passed to app as virtual key code)
`EV_WM_SIZE`	window size change
`EV_VN_xxxx`	an application-defined notification event sent by a document object to all its views

Response tables support inheritance; when you define a response table for a derived class, each entry overrides any corresponding base class response table entries. Of course, base class response tables entries that are not redefined in a derived class retain their original meaning.

3.2.4 The Doc/View model

In the *Document/View model* (or *Doc/View model* for short) the management of a window's data is treated as a separate task from the visual presentation of the data. It allows multiple views of different kinds on the same data. OWL supports this model, which fits the requirement for the structure editor, to have multiple views of various view types on a document, very well.

To use the Doc/View model, it is necessary to create a document class based on OWL's *TDocument* abstract base class. The primary function of the document class is to encapsulate the document, provide member functions that View objects can call to request data changes in a document and tell all associated views when data has been updated. In addition to this, the document class in the structure editor is responsible for storage and retrieval of documents to persistent storage (disk) and all Windows clipboard operations.

For each view type, it is necessary to create a separate view class based on the OWL abstract base class *TView*. Each view class displays the same data in a different manner or lets the user interact with it in a different way. A view forms an interface between a user interface object (often a window) and the document. The view class displays data from the document in its window and processes user input. When the user input is intended to change the document data, the view calls a corresponding document class member function to request this change. The document decides whether the data should be changed, and if so, broadcasts the change to all associated views.

For broadcasting document changes to all views, the document class uses the `NotifyViews` function which takes three parameters: an application-defined integer constant `EV_VN_xxxx` corresponding to the event and a `long` or `pointer` parameter to the event. Each View class that needs to respond to this notification should have a corresponding response table entry. For example, in the structure editor, the following view notifications exist:

`EV_VN_MIRRORSUBTREE`:	parameter `TNodeAddress *`. All views should create a mirror image of a subtree of the document tree, starting at the node indicated by the parameter.
`EV_VN_GLOBALFOCUS`:	parameter `TNodeAddress *`. All views should move the focus to the node indicated by the parameter.

In the structure editor, an MDI application, each view will appear in its own MDI child window.

Document objects are associated with View objects. The association is created through a template class, and you can create multiple instances of these templates. Each instance can specify details, such as the default directory, to search for the document files and what default file suffix to use with the documents.

Furthermore, the Doc/View model offers an application-wide document manager that maintains and coordinates document objects and corresponding view objects. It is an instance of *TDocManager* or a derived class which does the following:

- Manage the list of current documents and registered templates
- Handles the file menu (New, Open, Save etc.)

TDocManager supports both SDI and MDI applications.

4 C++ modelling of S-algebras

This chapter explains how C++ is used to model **S**-algebras. First, a way to model a signature with list operators **S** in C++ is presented. Then, to describe the editor part of the structure editor (where the actual term being edited resides), it is necessary to model a **S**-term algebra based on this signature. For the description of various types of view on the term, C++ models for a number of **S**-view algebras (all of which are based on the signature modelled earlier) are needed.

As explained in Section 2.10, a tree in the carrier set of the **S**-view algebra is very similar to the corresponding tree in the carrier set of the **S**-term algebra itself. Informally speaking, the only difference is that each node in the view tree carries an extra piece of information of type *I*. For each operator there is a function (element of *D*) which computes the associated value of type *I*, given the values for each of the operator's children.

For each view type, a separate **S**-view algebra is needed which is based on a view type-specific type *I* and a view type-specific set of functions *D*. Assume the editor has *v* different view types. The **S**-view algebra for view type *i*, 0**£***i*<*v* $\left(ViewTerm_{S,I_i,D_i}, F(I_i, D_i)\right)$ is based on a set I_i and a set of functions D_i specific to this view type.

The **S**-algebras used in the structure editor (a **S**-term algebra and a number of **S**-view algebras) are different but they have a common part: they are all based on the same signature **S**; this signature defines the structural properties of these **S**-algebras.

In this chapter, it is shown how the algebraic relationship between a single signature **S** and several **S**-algebras is reflected in the relationship between a single C++ model of the signature and the C++ models of several **S**-algebras. By defining the signature in terms of C++ template classes, I could reuse the C++ signature model for modelling both the **S**-term algebra and all **S**-view algebras.

The template classes have one parameter: the base class that should be used. The combination of an operator template class and a specific base class forms a class that can be instantiated to create an actual operator object.

The base classes used in the structure editor implementation are class hierarchies themselves, each level of which models different aspects of an operator object. Sections 4.2 and 4.3 contain all the details, but a brief overview of this hierarchy is given here.

The primary function of the base class is to model the connections between an operator object and its parent and each of its children. This makes it possible to connect a number of operator objects to form a tree structure (this amounts to modelling $Term_S$ or, equivalently, $Tree_S$). This is encapsulated in TTreeNode, the root of the base class hierarchy.

The secondary function is to model an operator object with member functions that return the kind of operator, its result sort, the operator argument sorts, pointers to its parent and children, etc. This is the purpose of the class TTerm, which is directly derived from TTreeNode. TTerm is the class to use as the operator class template argument when modelling a **S**-term algebra.

As pointed out before, a view tree is a term tree where each operator object carries an extra piece of information. When creating a C++ model of a **S**-view algebra for a view type *i*, the base class is called *view base class* (often referred to as $\mathbf{b}_i$ in this thesis). The view base class is the third layer of the base class hierarchy. $\mathbf{b}_i$ models the extra piece of information for view type *i* (set I_i,).

Illustration 4.1 in Section 4.1.2 shows the base class hierarchy and operator template classes.

In the C++ implementation, each view base class is associated with a collection of functions. This collection of functions matches the set D_i. associated with the **S**-view algebra for view type *i*.

This chapter first presents a C++ model for any signature **S** with list operators which consists of classes for each sort and each operator. Each operator class models the relevant properties of an operator: its result sort, its argument sorts, its number of arguments, whether it is a list or aggregate operator etc. It can be instantiated to form an operator object.

Then, given the model for a signature **S**, models for **S**-algebras (the **S**-term algebra and the **S**-view algebras) will be discussed. A C++ equivalent of the homomorphism from term to view algebra will also be introduced.

4.1 Modelling a signature with list operators in C++

This paragraph discusses the model model of a signature with list operators **S** (*S*,**G**), starting with the sorts.

4.1.1 Sort classes

For each sort, a C++ *template class* is defined which has the name of the base class as its parameter. This will come in handy later, when a similar class hierarchy to store view information is needed. By using a template, there is no need to duplicate Sort and Operator class declarations in the code for Views.

For each Sort in the signature, one abstract class is derived from the base class. In addition to this, the enumeration type 'Sort' defines a constant value `SORT_`*s* for each sort *s* and `SORT_NONE = 0`. The `GetSort` method returns the appropriate constant for each Sort class. The Sort classes provide a type-safe way to express the commonality of operators that have the same result sort.

With this structure, each sort $sort \in S$ will be translated to the following template class declaration:

```
template<class Base>
class      Tsort: public Base {
public:
   Sort                GetSort() { return SORT_sort; }
                       Tsort(int n = 0, int fixed = FALSE): Base(n, fixed) {}
};
```

Since the Sort classes do not provide an implementation of all pure virtual member functions of TTerm (see Section 4.2), they are abstract classes. The Sort class constructor does nothing: simply passes its arguments on to the base class constructor.

4.1.2 Operator classes

For each Operator in the signature, a template class is derived from its result sort template class. The enumeration type `Operator` defines a constant `OP_g` for each operator g $\in \cup\Gamma$ and `OP_NONE`. The operator classes are the ones that are actually instantiated by the structure editor when language constructs are added to the abstract syntax tree.

For the sake of structure editing, all hole operators have been included in **ÈG** (see 2.9), and a template class will exist for each hole. A hole of sort *s* is considered to be a constant operator with result sort *s*. Hole objects can be displayed as markers on the editor screen that indicate subtrees that are still missing from the term tree.

Similar to the Sort classes, the Operator classes are defined as *template classes*. The immediate base class of each operator class is its result sort class. The operator template parameter is passed on to the immediate base class; it is the name of the base class that sort class should use.

The recipe for creating a C++ operator class, given an aggregate operator definition $\mathtt{g}:\{f_1:s_1,\ldots,f_n:s_n\}\to s_0$, is shown below. Except `GetDesiredChildSort`, all member functions are defined inside the class declaration (this automatically makes them `inline` functions).

The function `GetChild` used in the operator class templates must be defined in `Base`. In the structure editor, Base must be a subclass of TTerm and `GetChild` is a member of TTerm.

```
template<class Base>
class      Tg: public Ts0<Base> {
public:
   Operator              GetOperator() { return OP_g; }

   TSorts1<Base>*        Getf1() { return (TSorts1<Base>*)GetChild(0); }
   …
   TSortsn<Base>*        Getfn() { return (TSortsn<Base>*)GetChild(n-1); }

   Sort                  GetDesiredChildSort(int c);
   int                   GetDesiredNumChildren() { return n; }

                         Tg(): TSorts0<Base>(n, TRUE) {};
};
```

As you can see above, the constructor `Tg()`simply passes on its arguments to the sort base class. The template function below defines the one member function not defined inside the class declaration. The reason for not defining this function inside the class declaration is that the compiler used for this project will not inline a function if it contains, among others, a `switch` statement. Using a separate declaration avoids the compiler warning message that would occur if this function were to be declared `inline`.

```
template<class Base>
Sort             Tg<Base>::GetDesiredChildSort(int c)
{
   switch(c) { case 0:       return SORT_s1;
                  ...
               case n-1:     return SORT_sn;
               default:      return SORT_NONE;
   }
}
```

The recipe for a list operator, such as $\mathtt{g}:\{f_1:s_1{}^*\}\to s_0$, is:

```
template<class Base>
class      Tg: public s0<Base> {
public:
   Operator              GetOperator() { return OP_g; }
   TSorts1<Base>*        Getf1(int c) { return (TSorts1<Base>*)GetChild(c); }

   Sort               GetDesiredChildSort(int c) {return c>=0?SORT_s1:SORT_NONE;}

   BOOL               IsList() { return TRUE; }

                      Tg(): TSorts0<Base>(0, FALSE) {}
};
```

For the example signature introduced in Section 2.2, the class hierarchy (without hole classes) is illustrated below. The base classes shown here, TTreeNode and TTerm, will be introduced in the next section.

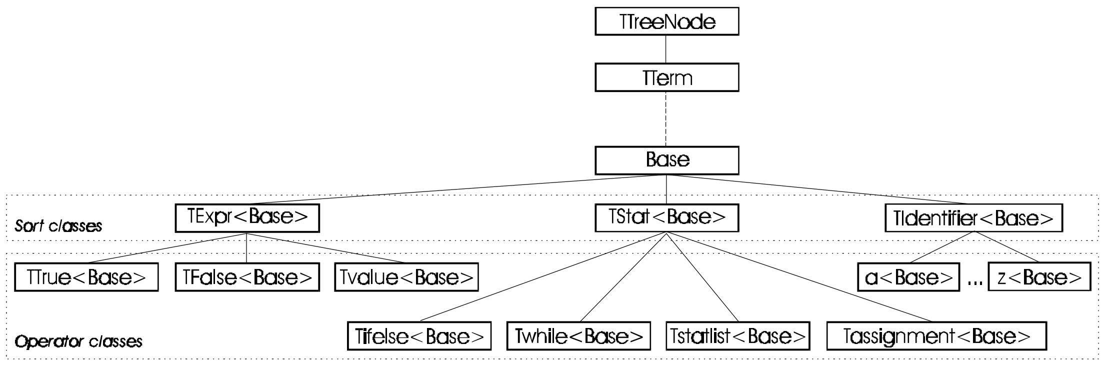


Figure 4.1 Class hierarchy for modelling the example signature

4.1.3 Sort and Operator Macros

To save a lot of typing, *macros* that contain the actual C++ sort and operator class template declarations can be defined. Assume that the sorts are numbered from 1 to $|S|$ in arbitrary order. Sort *i* is referred to as S_i . The operators are numbered from 1 to $|$**G**$|$ and operator *j* may be referred to as **G**$_j$., with the extra condition that operator number *k*, *k*£|S| is the hole of sort *k*: $\left(\forall k \in N : 1 \le k \le |S| : \Gamma_k = \Delta_{S_k}\right)$

The macros assume that an enumeration type Operator lists all operators identifiers according to the following pattern:

```
enum Operator {
   OP_NONE = 0,
   OP_S1HOLE, OP_S2HOLE, ... , OP_S|S|HOLE,
   OP_G|S|+1, OP_G|S|+2, ..., OP_G|G|
};
```

Given this, it is possible to define the sort classes S_i with a corresponding hole classes **G**$_i$ (for all 1£i£|S|) by using the following macro repeatedly:

```
DEFINE_SORT_AND_HOLE(Si)
```

This macro actually expands to:

```
DEFINE_SORT(Si)
DEFINE_OP0(SiHOLE, Si)
```

The macro `DEFINE_SORT` expands to a definition of the `SORT_Si` constant for this sort (it is set equal to the constant `OP_SiHOLE` from the `Operator` enumeration type), followed by the code fragment given in Section 4.1.1:

```
const Sort SORT_Si = OP_SiHOLE;
template<class Base>
class      TSi: public Base {
public:
   Sort                GetSort() { return SORT_Si; }
                       TSi(int n = 0, int fixed = FALSE): Base(n, fixed) {}
};
```

The `DEFINE_OP0` macro is a version of the recipe given in Section 4.1.2, for an operator of arity 0 (a constant).

Now that sorts and holes have been defined, I will move on to the 'real' operators in the signature.

An aggregate operator, not a hole, of arity *n*, $g:\{f_1:s_1,\ldots,f_n:s_n\} \rightarrow s_0$ is defined as follows:

```
DEFINE_OPn(g, s0, f1, s1, ..., fn, sn)
```

This macro expands to a version for an operator of arity *n* of the recipe given in Section 4.1.2, which will not be repeated here. Obviously, for each operator arity *n* that occurs in the signature, a corresponding `DEFINE_OPn` macro must be defined.

There is macro for list operators too. A list operator $g:\{f_1:s_1^*\} \rightarrow s_0$ is defined by the following macro:

```
DEFINE_OP_LIST(g, s0, f1, s1)
```

In addition to this, the following definitions are needed by other parts of the structure editor:

```
const Sort    TERMSORT = SORT_s;
```

where *s* is the sort of the term to be edited in the structure editor. Finally, the following definitions serve to separate the holes from the 'real operators':

```
const Operator   OP_FIRSTHOLE    = OP_G1;
const Operator   OP_LASTHOLE     = OP_G|S|;
const Operator   OP_FIRST       = OP_G|S|+1;
const Operator   OP_LAST        = OP_G|G|;
```

Using the macros, the C++ equivalent of the labeled example signature (with holes) is:

```
enum Operator {
   OP_NONE,
   OP_StatHOLE, OP_ExprHOLE, OP_IndentifierHOLE,
   OP_ifelse, OP_while, OP_statlist, OP_assignment,
   OP_a, OP_b, ... , OP_z, OP_True, OP_False, OP_value
};

DEFINE_SORT_AND_HOLE(Stat)
DEFINE_SORT_AND_HOLE(Expr)
DEFINE_SORT_AND_HOLE(Identifier)

DEFINE_OP2(assignment, Stat, dest, Identifier, source, Expr)
DEFINE_OP3(ifelse, Stat, guard, Expr, s_true, Stat, s_false, Stat)
DEFINE_OP2(while, Stat, guard, Expr, body, Stat)
DEFINE_OP_LIST(statlist, Stat, s, Stat)

DEFINE_OP0(a, Identifier)
DEFINE_OP0(b, Identifier)
...
DEFINE_OP0(z, Identifier)

DEFINE_OP1(value, Expr, source, Identifier)

DEFINE_OP0(False, Expr)
DEFINE_OP0(True, Expr)

const Sort    TERMSORT = SORT_Stat; // terms in the editor are of sort Stat

const Operator   OP_FIRSTHOLE   = OP_StatHOLE;
const Operator   OP_LASTHOLE    = OP_IdentifierHOLE;
const Operator   OP_FIRST       = OP_ifelse;
const Operator   OP_LAST        = OP_value;
```

4.2 S-term algebra modelling

In Chapter 2, we have seen two ways to look at a term over a signature **S** (*S*,**G**): as a term (element of *Term*$_{\mathbf{S}}$) and as a tree (a function from a tree domain to **ÈG**, element of *Tree*$_{\mathbf{S}}$). For an efficient model of a **S**-term algebra in C++, a third variation of this theme is needed. This is a term tree representation that consists of a number of operator objects linked by pointers.

Each operator object in the pointer-based term tree is an instance of the corresponding operator class and it has direct pointers to its parent and each of its children. To make these pointers part of each operator object, one possibility is to include the pointer members in the operator class directly. A cleaner approach, however, is to handle all tree-related issues in separate tree classes.

4.2.1 C++ tree classes

I will now present the two tree classes used in the structure editor project. The first, TTreeNode, encapsulates the information necessary to link a single tree node to its parent and children. The second is TTree, which is a template class that encapsulates an entire tree; it offers a wide variety of operations to construct and manipulate the tree.

TTreeNode

TTreeNode has the following data members, which allow efficient implementation of all tree manipulations needed by the structure editor:

- a pointer to the parent node
- an array of pointers to all child nodes.
- a child number: this is the object's index in its parent's array of pointers to children.

TTreeNode is required to be a base class of all objects that need to be in a tree structure managed by TTree.

```
class TTreeNode {
public:
   inline                    TTreeNode(int n = 0, int fixed = FALSE);
   virtual                  ~TTreeNode() {}
   inline TTreeNode         *GetChild(int n) const;
   inline TTreeNode         *GetParent() const;
   inline int                GetNumChildren() const;
   void                      SetChildNumbers(int n);
   long                      CountSubtreeNodes() const;
   BOOL                      IsDescendant(TTreeNode *anc) const;

   TTreeNode                *Parent;
   int                       MyChildNumber;
   TArrayAsVector<TTreeNode*> Children;
};

inline                       TTreeNode::TTreeNode(int n, int fixed):
                             Children(n,0,fixed?0:2),Parent(0),MyChildNumber(0)
{
}
```

The class TTreeNode holds the data members listed above, where the array of pointers to children is implemented using Borland's TArrayAsVector container class.

The constructor takes two parameters: an integer *n* stating the initial size of the `Children` pointer array and a Boolean indicating whether the tree node has a fixed or variable number of children. The `Parent` and `MyChildNumber` members are initialized to 0.

The destructor is declared virtual so that even if a tree node is deleted using a pointer to TTreeNode or another base class (TTree is likely to do this), the correct destructor corresponding to the tree node's actual type is called.

The class TTreeNode has the following public member functions:

```
TTreeNode    *GetChild(int n) const;
```

If 0£n<#children, this function returns a pointer to a node's n'th child. Otherwise it returns 0.

```
TTreeNode    *GetParent() const;
```

Returns the value of the `Parent` data member.

```
int           GetNumChildren() const;
```

Returns a node's current number of children.

```
void          SetChildNumbers(int n);
```

Sets all child node's `MyChildNumber` members, starting with child *n*.

```
long          CountSubtreeNodes const;
```

Counts the number of descendants, including this node (so the return value is at least 1).

```
BOOL          IsDescendant(TTreeNode *anc) const;
```

Checks whether `node` is a descendant of the node pointed to by `anc` (ancestor).

TTree template class

I aimed to develop a tree class that can be reused whenever a tree type data structure occurs in this project. It is called TTree and it is general template container class. Whenever a TTree object is constructed, the type of the tree's nodes must be given as the template argument. The node type is required to be a subclass of TTreeNode because TTree uses TTreeNode to link a node to its parent and children.

```
template<class T>
class TTree {
public:
   inline              TTree():Root(0),NumNodes(0) {}
                      ~TTree();

   inline   T         *GetRoot() const;
   inline   long       GetNumNodes() const;
   inline   BOOL       IsLeaf(T *e) const;
```

```
      inline   BOOL        IsRoot(T *e) const;

      inline   int         GetMyChildNumber(T *e) const;
      inline   T          *GetParent(T *e)   const;
      inline   T          *GetChild(T *e, int n) const;
      inline   int         GetNumChildren(T *e) const;
      inline   T          *GetSibling(T *e, int n) const;
      inline   T          *GetLeftSibling(T *e) const;
      inline   T          *GetRightSibling(T *e) const;

      T                   *GetLastNode() const;
      T                   *GetNextNode(T *e) const;
      T                   *GetPrevNode(T *e) const;

      inline   void        InsertChild(T *par,  int childnumber, T *child);
      void                 AddChild(T *par, T *child);
      void                 ReplaceChild(T *par, int childnumber, T *newchild, BOOL del=TRUE);
      void                 RemoveChild(T *par, int childnumber, BOOL del=TRUE);

      inline   void        RemoveSubtree(T *subtree, BOOL del=TRUE);
      inline   void        ReplaceSubtree(T *subtree, T *replacement, BOOL del=TRUE);

      void                 GetNodeAddress(T *node, TNodeAddress& address) const;
      T                   *NodeAt(const TNodeAddress& address) const;
   private:
      void                 DoRecursive(T *e, void pre(T *), void post(T *));

      T                   *Root;
      long                 NumNodes;
   };
```

As private data members, the class TTree holds a pointer to the root TTreeNode and the number of nodes the tree currently has. The constructor creates an empty tree (`Root`=0, `NumNodes` = 0) and the destructor traverses the tree and deletes all elements.

The class TTree has the following public member functions:

```
T*          GetRoot();
```
Returns a pointer to the root node, or 0 if the tree is empty.

```
long        GetNumNodes();
```
Returns the number of nodes in the tree.

```
BOOL        IsLeaf(T* e);
```
Returns TRUE if the node pointed to by e is a leaf, FALSE if not.

```
BOOL        IsRoot(T* e);
```
Returns TRUE if the node pointed to by e is the root node of the tree.

```
int         GetMyChildNumber(T* e);
```
Returns the index of e in the parent's array of children.

```
int         GetNumChildren(T* e);
```
Returns the number of children of the node pointed to by e. If e is 0, this function returns 1 if the tree has a root (it is nonempty), otherwise 0.

```
T          *GetChild(T* e, int n);
```
Returns the `n`'th child of the node pointed to by e. Child #0 of 0 (null pointer) is the root of the tree. If `n` is an invalid child number, this function returns 0.

```
T          *GetSibling(T* e, int n);
```
Returns a sibling of the node pointed to by e (the n'th child of the parent of e). If n is an invalid child number, this function returns 0.

```
T          *GetLeftSibling(T* e);
```
Returns the sibling immediately to the left of the node pointed to by e. If e has no left sibling, this function returns 0.

```
T          *GetRightSibling(T* e);
```
Returns the sibling immediately to the right of the node pointed to by e. If e has no right sibling, this function returns 0.

```
T          *GetParent(T* e);
```
Returns a pointer to the parent of the node pointed to by e. If e is the root node, this function returns 0.

```
T          *GetNextNode(T* e);
```
Returns a pointer to the node that would follow this node in a pre-order tree traversal. If e is the last node in the tree, this function returns 0.

```
T          *GetPrevNode(T* e);
```

Returns a pointer to the node that would precede this node in a pre-order tree traversal. If e is the first node in the tree (the root node), this function returns 0.

```
T          *GetLastNode();
```

Returns a pointer to the node that would be the last node visited in a pre-order traversal of this tree.

```
void        InsertChild(T* par,  int childnumber, T* child);
```

Expands the children array of the node pointed to by `par` and inserts `child` into it, at position `childnumber`. `child` is allowed to have children itself, so an entire subtree can be inserted with this function.

```
void        AddChild(T* par, T* child);
```

Expands the children array of the node pointed to by `par` and add `child` at the end. `child` is allowed to have children itself, so an entire subtree can be added with this function.

```
void        ReplaceChild(T* par, int childnumber, T* newchild, BOOL del=TRUE);
```

Replaces child number `childnumber` of node `par` by the node pointed to by `child` at the end. If del is TRUE ,the existing subtree will be deleted. If not, it will only be detached. Again, `child` is allowed to have children itself, so an entire subtree can be replaced with this function.

```
void        RemoveChild(T* par, int childnumber, BOOL del=TRUE);
```

Removes child number `childnumber` of node `par`. If `del` is TRUE the child and all its descendants will be deleted. If not, they will only be detached.

```
void        RemoveSubtree(T* subtree, BOOL del=TRUE);
```

Removes the node pointed to by `subtree` and all its descendants from the tree. If `del` is TRUE, the node and its descendants will be deleted. If not, they will only be detached.

```
void        ReplaceSubtree(T* subtree, T* replacement, BOOL del=TRUE);
```

Replaces the subtree pointed to by `subtree` by the subtree pointed to by `replacement`. If `del` is TRUE, the node pointed to by `subtree` and its descendants will all be deleted. If not, they will only be detached.

```
void        GetNodeAddress(T *node, TNodeAddress& address) const;
```

Stores node address of `node` in the TNodeAddress object `address`.

```
T          *NodeAt(const TNodeAddress& address) const;
```

Returns pointer to node addressed by the TNodeAddress object `address`.

TNodeAddress class

The TNodeAddress class models the concept of a tree node address, defined in Chapter 2, definition 2.13. It allows memory location-independent addressing of tree nodes.

```
class TNodeAddress: public TArrayAsVector<short> {
public:
   inline          TNodeAddress() : TArrayAsVector<short>(9,0,10) {}

   inline int      Length() const;
   inline short    GetChildNumber(int n) const;
};
```

A tree node address is implemented as a vector of `shorts`. Its initial size is 10 and, when more space is required (this occurs when the depth of a node exceeds the initial size), the array is incremented in steps of 10. This class is used by the `GetNodeAddress` and `NodeAt` member functions of TTree.

```
inline int       Length() const;
```

Returns the length of the node address sequence.

```
inline short     GetChildNumber(int n) const;
```

Gets the `n`'th element of the node address sequence (0 indicates the first element).

An example code fragment that builds a tree consisting of a few nodes and does some manipulations:

```
void f()
{
   TTree<TTreeNode>      tree;

   TTreeNode            *n1, *n2, *n3, *n4, *n5;
   TNodeAddress          a;

   n1 = new TTreeNode; n2 = new TTreeNode; n3 = new TTreeNode;
   n4 = new TTreeNode; n5 = new TTreeNode;
   tree.AddChild(0, n1);            // Set root node;
   tree.AddChild(n1, n2);           // Add child to root node
   tree.AddChild(n1, n3);           // Add another child to root node
   tree.AddChild(n2, n4);           // Add child to n2
   tree.AddChild(n2, n5);           // Add another child to n2

   tree.RemoveSubtree(n2, FALSE);   // remove subtree, do not delete nodes
   tree.AddChild(n3, n2);           // attach subtree to n3.

   BOOL b = n5->IsDescendant(n2)    // b == TRUE;

   tree.GetNodeAddress(n5, a);      // node address of n5 is <0, 0, 1>

   // tree goes out of scope, destructor deletes tree nodes n1, n2 and n3.
}
```

4.2.2 TTerm class

The class TTerm is an abstract class, derived from TTreeNode, and it adds virtual member functions to retrieve a node's sort (GetSort), its operator (GetOperator), the desired sorts for its children, an access function to retrieve a pointer to a child node, streaming support etc. TTerm is intended to be used as the direct base class of all nodes in the term tree representation and hence, it should be used as the Base parameter to the operator template classes. TTerm also serves as indirect base class to nodes in the view tree representations.

The example signature has constant operators named *a* ... *z* to model identifiers. However, with identifier names that can be of arbitrary length, this will yield an unbounded number of operators. Instead of defining a separate operator for each identifier, a single operator, *identifier*, is defined. To distinguish different identifiers, each identifier object has associated information which contains the name of the identifier. The TTerm class has `SetData` and `GetData` member functions to set and access the operator information.

```
class     TTerm: public TTreeNode {
public:
   inline            TTerm(int n=0, int fixed=FALSE): TTreeNode(n, fixed),Data(0),Length(0){}
                    ~TTerm();

   virtual Operator  GetOperator() = 0;
   inline char      *GetOpName()               { return ::GetOpName(GetOperator()); }
   virtual int       GetDesiredNumChildren()   { return 0; }
   virtual BOOL      IsList()                  { return FALSE; }

   virtual Sort      GetSort() = 0;
   inline char      *GetSortName()             { return ::GetSortName(GetSort()); }

   TTerm            *GetChild(int n)           { return (TTerm*)TTreeNode::GetChild(n);}
   virtual Sort      GetDesiredChildSort(int n) { return SORT_NONE; }

   void              SetData(void *data, int len);
   void             *GetData()                 { return Data; }
   int               GetDataLength()           { return Length; }

   int               GetTermLength();
   friend ostream&   operator<<(ostream&, TTerm *);
private:
   void             *Data;
   int               Length;
};
```

The constructor takes two parameters: the integer *n* is the initial number of children of this node and the Boolean *fixed* indicates whether the tree node has a fixed or variable number of children.

These two values are passed on to the TTreeNode constructor. The `Data` and `Length` members are initialized to 0.

If this operator has associated data, the destructor frees its memory.

The class TTerm has the following public member functions:

```
virtual Operator     GetOperator() = 0;
```
This pure virtual function must be defined by the **g** operator class to return the correct `OP_g` constant.

```
inline char         *GetOpName();
```
This function uses `GetOperator()` to find the operator; then returns a string in the operator name array.

```
virtual int          GetDesiredNumChildren();
```
This virtual function (which returns 0) must be overridden by an operator class if the operator's arity is greater than 0. `GetArity` would have been a better name for this function.

```
virtual BOOL         IsList();
```
This virtual function (which returns FALSE) must be overridden by an operator class to return TRUE if it is a list operator.

```
virtual Sort         GetSort() = 0;
```
This pure virtual function must be defined by the sort class of sort *s* to return the correct `SORT_s` constant.

```
inline char         *GetSortName();
```
This function uses `GetSort()` to find the sort; it then returns the corresponding string in the sort name array.

```
TTerm               *GetChild(int n);
```
This function uses calls `TTreeNode::GetChild` and casts the result to `TTerm *`.

```
virtual Sort         GetDesiredChildSort(int n);
```
This virtual function, which by default returns `SORT_NONE`, must be overridden by an operator class if the operator is not a constant. The purpose of the function is to enable an application to retrieve the argument sorts for this operator.

```
void                 SetData(void *data, int len);
```
Sets operator information. A memory block of length `len` is allocated, and the memory pointed to by `data` is copied into it.

```
void                *GetData();
```
Gets operator information.

```
int                  GetDataLength();
```
Gets length of operator information.

```
int                  GetTermLength();
```
Get number of characters needed to stream this term.

```
friend ostream&      operator<<(ostream&, TTerm *);
```
Stream term in following format: operator name { operator info, if any } [child 0, ..., child n-1]

4.2.3 Term representation

At this point, all the ingredients to model a **S**-term algebra have been presented:

- A C++ model for the signature **S**
- A tree container class
- The TTerm class

It is now shown how to combine these ingredients in a C++ program that models the **S**-term algebra of the running example signature. The program builds and prints the example term given in Section 2.4:

ifelse[value[a],assignment[b,False],assignment[c,True]]

The C++ code that follows is based on the C++ signature definition given in Section 4.1.3:

```
void f()
{
   TTree<TTerm>         TermTree;

   TTerm               *ifelse = new Tifelse<TTerm>;
   TTerm               *value  = new Tvalue<TTerm>;
   TTerm               *a      = new Ta<TTerm>;
   TTerm               *assign1= new Tassignment<TTerm>;
   TTerm               *b      = new Tb<TTerm>;
   TTerm               *False  = new TFalse<TTerm>;
   TTerm               *assign2= new Tassignment<TTerm>;
   TTerm               *c      = new Tc<TTerm>;
   TTerm               *True   = new TTrue<TTerm>;

   TermTree.AddChild(0,       ifelse);
   TermTree.AddChild(ifelse,  value);
   TermTree.AddChild(value,   a);
   TermTree.AddChild(ifelse,  assign1);
   TermTree.AddChild(assign1, b);
   TermTree.AddChild(assign1, False);
   TermTree.AddChild(ifelse,  assign2);
   TermTree.AddChild(assign2, c);
   TermTree.AddChild(assign2, True);

   cout << TermTree.GetRoot();                  // TermTree.GetRoot() == ifelse
   // outputs 'ifelse[value[a],assignment[b,False],assignment[c,True]]'
   int tl  = TermTree.GetRoot()->GetTermLength(); // tl==55, length of the above output stream

   cout << assign2;
   // outputs 'assignment[c,True]'

   int nc   = ifelse->GetNumChildren();        // nc == 3
   int dnc  = ifelse->GetDesiredNumChildren(); // dnc == 3
   Sort s0  = ifelse->GetDesiredChildSort(0);  // s0 == SORT_Expr
   Sort s1  = ifelse->GetDesiredChildSort(1);  // s1 == SORT_Stat
   Sort s2  = ifelse->GetDesiredChildSort(2);  // s2 == SORT_Stat

   TTerm *t = TermTree.GetLeftSibling(assign2);// t == assign1
}
```

4.3 S-view algebra representation

The intention is to create a design for the structure editor that neatly separates the abstract syntax tree document storage and manipulation from document display and user interaction. This makes it possible to have multiple types of *views* on the same document, where each type of view defines a particular way of displaying data and letting the user interact with it.

The class that encapsulates the abstract syntax tree (the *TTermEditor* class, introduced in Chapter 5) is unaware of the number and types of views on it, so that new view classes can be designed and added to the structure editor without having to change the document class.

Each view type is modelled as a different **S**-view algebra. Conceptually, for view type *i*, the **S**-view algebra corresponds to the **S**-term algebra plus, for each term tree node (operator object), an extra piece of information of type I_i. The following section discusses basic view features for the structure editor and a data type (TViewNode) that supports these features. If a view type *i* needs additional per-node information, it can derive a more elaborate I_i from TViewNode.

4.3.1 Basic view features

The graphical representation of a node is a rectangle. There can be an arbitrary number of instances of a node's graphical representation within its parent rectangle. It is assumed that a node representation's bounding rectangle completely encloses all representations of its children. The illustration below shows an acceptable mapping from the term tree of the example term

ifelse[value[a],assignment[b,False],assignment[c,True]]

to a collection of bounding rectangles:

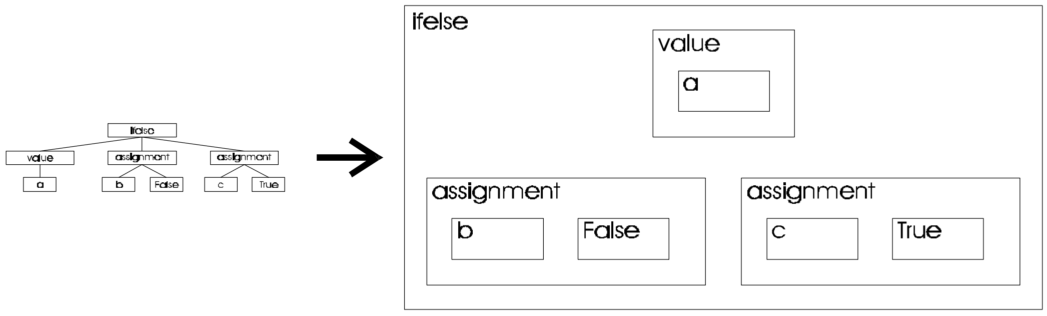


Figure 4.2 Bounding rectangles in graphical term representation

Given a certain (x, y) coordinate, the view must be able to determine efficiently to which node rectangle, if any, it refers. This is useful for processing mouse clicks in the structure editor application.

A node can be ‘folded’, which means that it and all its descendants disappear from the View display. A marker will be shown to indicate the presence of a folded subtree. The unfold operation will cause a folded node and its hidden descendants to reappear on the screen.

4.3.2 View nodes

View tree nodes use the same Operator template classes as nodes of the document tree. This means that there is no need to redeclare the entire signature for each view type. These template classes have a parameter to allow specification of the base class to use. Instead of TTerm (used for modelling the term tree), a view-specific base class $\mathbf{b}_i$ is used.

A base class for view tree nodes is derived from TTerm (which, in turn, is derived from the TTreeNode), and has data members to carry per-node information needed for supporting the basic view features indicated above. The class TViewNode contains the following information about a node: a list of top left corner coordinates for each instance in the display, the node bounding rectangle size and a Boolean indicating whether it is folded or not. If a view needs more information, its node base class will be a subclass of TViewNode with additional data members.

The C++ declaration of the TViewNode class:

```
class   TViewNode : public TTerm {
public:
                        TViewNode(int n = 0, int fixed = FALSE):TTerm(n, fixed),
                                       Folded(FALSE),   NodePositions(0,0,2) {}

   inline TViewNode    *GetChild(int n) const;
   inline TViewNode    *GetParent() const;

   inline BOOL          GetFolded() const;
   inline void          SetFolded(BOOL folded);
   inline TSize&        NodeSize();
   inline int           NodeHeight() const;
   inline int           NodeWidth() const;


   virtual void         Draw(TMyDC&, TPoint&) = 0;
   virtual void         CalcSize(TMyDC&)      = 0;

   void                 SetNodeSize(const TSize& sz);
   void                 AddNodePosition(const TPoint& pt);
   TPoint&              GetNodePosition(int n) const;
   int                  GetNumNodePositions() const;
   TViewNode           *WhichNode(const TPoint& point);
   BOOL                 IsVisible() const;

   void                 FlushChildrenNodePositions();
private:
   TSize                Size;
   BOOL                 Folded;
   TArrayAsVector<TPoint>   NodePositions;
};
```

The TViewNode class is directly derived from the term basic class, TTerm. As private data members it has a TSize, which holds the size of the bounding rectangle of this node's graphical representation, an array of node positions (relative to the origin of the view window coordinate system), one for each copy of this node on the screen and a Boolean indicating whether the node is folded or not.

The constructor initializes the node position array to have an initial size of 1 and a growth delta of 2. In addition, it sets folded to FALSE and passes its arguments on to the TTerm constructor. TViewNode has the following public member functions:

`TViewNode     *GetChild(int n) const`

Calls `TTerm::GetChild` and casts the result from `TTerm *` to `TViewNode *`.

`TViewNode     *GetParent() const`

Calls `TTerm::GetParent` and casts the result from `TTerm *` to `TViewNode *`.

`BOOL          GetFolded() const`

Returns the Boolean value that indicates whether this node is currently folded or not.

`void          SetFolded(BOOL folded)`

Sets the Boolean value that whether this node is currently folded or not to folded.

`TSize&        NodeSize()`

Returns a reference to the node's TSize object.

`inline int    NodeWidth() const;`

Returns the node's width (`Size.cx`).

`inline int    NodeHeight() const;`

Returns the node's height (`Size.cy`).

`void          SetNodeSize(TSize& sz);`

Sets the size of the node by setting `Size` to `sz` and removes all elements from the `NodePositions` array. This function is commonly called by the CalcSize() member function.

`void          AddNodePosition(TPoint& pt);`

Adds the position given by `pt` to the `NodePositions` array. The `Draw()` member function has a screen coordinate as one of its arguments, and calls this function to add the coordinate to the node positions array. Thus, if a single node is drawn 3 times at different screen locations, the `NodePositions` array will hold all 3 locations.

`TPoint&       GetNodePosition(int n) const;`

Returns the position of the n'th graphical representation of this node on the screen, where 0£n<`GetNumNodePositions()`.

`int           GetNumNodePosition() const;`

Returns the number of graphical representations of this node on the screen (the number of `TPoint` objects in the `NodePositions` array).

`void          CalcSize(TMyDC& dc)`

This is a pure virtual function and it needs to be defined in a derived class. Its job is to calculate the size of this node's bounding rectangle, possibly using font size and other information from the display context dc. As the size often depends on the sizes of the child nodes' bounding rectangles, CalcSize recursively calls itself for each child node. When it has computed the node size, it must call `SetNodeSize()` to store this information in the node object.

`void          Draw(TMyDC& dc, TPoint& pt)`

This function adds the position of the node graphical representation it is about to draw by calling `AddNodePosition(pt)`. It then paints the node to the given display context `dc` at position `pt`. For each child, Draw() computes one (or sometimes more than one) position and then calls itself recursively to paint the child node at the computed position(s). Before painting any children, it must call `FlushChildrenNodePositions()`.

`TViewNode     *WhichNode(TPoint& point)`

This function checks whether `point` is in one of this node's bounding rectangles (using all points in the `NodePositions` array and the `Size` member). If not, it returns 0 right away. Otherwise, it calls itself recursively for each of its children to see if the point is inside one of them and WhichNode returns a nonzero value. If this is the case, this value is returned. If the point was outside all children (but inside this node), WhichNode returns the **this** pointer.

```
BOOL          *IsVisible() const;
```

This function returns FALSE when this node is invisible because it is folded itself or at least one of its ancestors is folded.

```
void           FlushChildrenNodePositions();
```

This function empties the `NodePositions` array for each of this node's children.

4.3.3 View trees

The class TViewTree encapsulates view trees. It is a template class with one argument - the view base class. Its constructor models part of the homomorphism from term tree to view tree; it creates the view tree that corresponds to the term tree specified by its `TTree<TTerm> *` argument. As the example in Section 4.3.5 will show, calling `CalcSize` for the view tree root node completes computation of the homomorphism.

```
template<class T>
class   TViewTree: public TTree<T> {
public:
                         TViewTree(TTree<TTerm> *termtree);
   inline T             *GetFocus() const;
   inline void           SetFocus(T *);
   inline T             *WhichNode(TPoint& point) const;
   void                  CreateMirror(T *parent, int childnumber, TTerm *term);
private:
   T                    *Focus;
};
```

The class is derived from the basic tree class, TTree. As a private data member, the class TViewTree holds a pointer to the focus node. The constructor takes a pointer to a `TTree<TTerm>`, which is the type of the term tree, as an argument. It builds the mirror view tree of type `TTree<T>` by calling `CreateMirror(0, 0,` *root of document tree*`)`.

```
void          CreateMirror(T *parent, int cn, TTerm *term)
```

Creates a mirror node of the node pointed to by term and adds it as a child number `cn` to the node pointed to by parent. Recursively calls `CreateMirror` for each of this term node's children. The code below uses the template function `CreateOperatorNode` introduced in Section 5.1. Given an operator constant `OP_`*g* for an operator *g*, this function creates the corresponding operator object `T`*g*`<Base>` and returns a pointer to it.

```
template<class T>
void TViewTree<T>::CreateMirror(T *parent, int childnumber, TTerm *term)
{
   T  *node = CreateOperatorNode((T *)0, term->GetOperator());

   if(node) {
      node->SetData(term->GetData(), term->GetDataLength());
      if(childnumber < GetNumChildren(parent)) {
         ReplaceChild(parent, childnumber, node);
      } else if(childnumber == GetNumChildren(parent)) {
         AddChild(parent, node);
      }
      int nc = term->GetNumChildren();
      for(int i = 0; i<nc; i++) {
         CreateMirror(node, i, term->GetChild(i));
      }
   }
}
```

TTreeView<T> has the following public member functions:

```
T         *GetFocus() const
```

Retrieve the pointer to the focus node.

```
void       SetFocus(T *) const
```

Sets the pointer to the focus node.

```
T         *WhichNode(TPoint& point) const
```

Find the node indicated by a mouse click at the given point (coordinate) in the view window. The assumption here is that a node's bounding rectangle completely contains the bounding rectangles of all of its children. This implies that when the point is outside a node's bounding rectangle, it is also outside all of its descendants' bounding rectangles. The node indicated by the given point is the 'deepest' node that still contains it. If no node contains the point, the function returns 0.

4.3.4 Modelling the functions D

For view *i*, the set I_i is modelled by a corresponding view base class, **b**ᵢ. Each view algebra also has a set of functions $D_i = \left\{ d_{i,\mathrm{g}} \middle| \mathrm{g} \in \cup\Gamma \right\}$ (see Section 2.10).

For view type *i*, the purpose of function $d_{i,g}$ is to compute the I_i of an operator object *g*, given the I_i objects of its children.

The C++ equivalent of each function $d_{i,\mathbf{g}}$ for view type *i* and operator *g* is:

`void   T`*g*`<`*bᵢ*`>::CalcSize(TMyDC& dc)`

where $\mathbf{b}_i$ is a class, derived from TViewNode, which models the I_i. $\mathbf{b}_i$ is called the *view base class* for view type *i*.

For each operator and view type, a separate version of this function must be defined. Each function expresses unique (graphical) characteristics of an operator / view type combination.

Since all $\mathbf{b}_i$'s are derived from TViewNode, they will always contain the node bounding box size (`TViewNode::Size`). So, the least each `CalcSize` function must do is calculate the size of the node bounding rectangle, given the sizes of the bounding rectangles of its children (hence its name). If I_i is a derived class of TViewNode with additional data members, more processing will probably be necessary. Each operator's `CalcSize` member function relies on the **b** values of the operator's children, so the `CalcSize` functions must execute in a bottom-up fashion: it is the responsibility of each `CalcSize` function to call the `CalcSize` member function of each child node before doing its own calculations. To facilitate these calculations, which are often graphics-related, the parameter `dc` provides a reference to a TMyDC display context object. In short, a `CalcSize` function for an aggregate operator $\mathrm{g}{:}\{ f_1 : s_1, \ldots, f_n : s_n \} \to s_0$ follows this pattern:

```
void    Tg<b1>::CalcSize(TMyDC& dc)
{
   TSorts1<b1>   *child1 = Getf1(); child1->CalcSize(dc);
   ...
   TSortsn<b1>   *childn = Getfn(); childn->CalcSize(dc);

   SetNodeSize( calculate node size based on child1...childn->NodeSize()  );
}
```

And for a list operator, $\mathrm{g}{:}\{ f_1{:}s_1^{*} \} \to s_0$, this is a likely pattern:

```
void    Tg<b1>::CalcSize(TMyDC& dc)
{
   TSize      sz = some initial value;

   for(int j = 0; j<GetNumChildren(); j++) {
      TSorts1<b1>   *child = Getf1(j); child->CalcSize(dc);

      make an adjustment of size sz based on child->NodeSize()
   }
   SetNodeSize(sz);
}
```

Thus, to calculate the I_i of all nodes in the view tree for view type *i*, it suffices to call `CalcSize` for the root node.

When all node information I_i is computed, the `Draw` member function can be used. A `Draw` function draws a node in the view window at a given position. Using the given position and the sizes of its child nodes, for each child node, the `Draw` function computes an arbitrary number of positions (usually 1) and calls itself recursively. Thus, to draw the entire view tree, it suffices to call `Draw` for the view tree root node. A `Draw` function for an aggregate operator g:$\{ f_1 : s_1 , \dots , f_n : s_n \} \to s_0$ in view type *i* with view base class ***bi*** can be defined according to the following pattern:

```
void    Tg<bi>::Draw(TMyDC& dc, TPoint& pos)
{
   AddNodePosition(pos);
   FlushChildrenNodePositions();

   TSorts1<bi>   *child1 = Getf1();
   ...
   TSortsn<bi>   *childn = Getfn();

   dc.XXX // Draw some decorations

   < compute position of child x>
   childx->Draw(dc, <computed position of child x>);

   dc.XXX // Draw decorations

   < compute position of child y>
   childy->Draw(dc, <computed position of child y>);

   ... and so on (child nodes can be drawn in any order, any number of times (incl. 0))


   dc.XXX // Draw decorations


}
```

And for a list operator, g:$\{f_1{:}s_1^*\} \to s_0$, it will probably look something like this:

```
void    Tg<bi>::Draw(TMyDC& dc, TPoint& pos)
{
   AddNodePosition(pos);
   FlushChildrenNodePositions();

   dc.XXX // Draw some decorations

   for(int j = 0; j<GetNumChildren(); j++) {
      TSorts1<bi>   *child = Getf1(j);

      < compute position of child j>
      child->Draw(dc, <computed position of child >);

      dc.XXX // Draw decorations
   }

   dc.XXX // Draw more decorations
}
```

This function's body contains graphics calls (indicated by `dc.XXX` here) to draw the **g** node. It is every Draw function's responsibility to call itself recursively for its children when they need to be displayed.

Obviously, the total number of `CalcSize` and `Draw` member functions to implement for a signature **S** (S, **G**) will be the number of operators |**ÈG**| times the number of view types.

4.3.5 C++ example

This section presents a small, but fairly complete **S**-view algebra example. Again, assume that the signature has been defined using the code fragment in Section 4.1.3. Not all `CalcSize` and `Draw` functions for the signature are implemented, but only the ones for the operators actually used in the term tree built by the example program.

assignment[a, True]

The graphical output produced will be **a := True**.

```
extern TMyDC      dc;

void f()
{
   TTree<TTerm>       TermTree;

   TTerm               *assign = new Tassignment<TTerm>;
   TTerm               *a      = new Ta<TTerm>;
   TTerm               *True   = new TTrue<TTerm>;

   TermTree.AddChild(0,      assign);
   TermTree.AddChild(assign, a);
   TermTree.AddChild(assign, True);

   // TermTree now holds the term assign[a,True]


   TViewTree<TViewNode>   ViewTree(TermTree);

   // TViewTree constructor has created a view tree corresponding to the term tree
   // View tree is based on standard base class (TViewNode), nothing fancy here.

   ViewTree.GetRoot()->CalcSize(dc);
   // Do CalcSize on entire view tree, which completes the homomorphism: ViewTree = hStat(TermTree)

   ViewTree.GetRoot()->Draw(dc, TPoint(20, 20));
   // Draw view tree, top left edge at (20, 20).

   // TermTree will go out of scope and will delete nodes pointed to by a, assign and True.
   // ViewTree will also clean up after itself.
}

// Implementation of CalcSize() and Draw() functions for assignment, a and True operators:

void    Ta<TViewNode>::CalcSize(TMyDC& dc)
{  SetNodeSize(dc.GetTextExtent(“a”, 1));
}
void    Ta<TViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  AddNodePosition(pos);
   FlushChildrenNodePositions();
   dc.TextOut(pos, “a”);
}


void    TTrue<TViewNode>::CalcSize(TMyDC& dc)
{  SetNodeSize(dc.GetTextExtent(“True”, 4));
}
void    TTrue<TViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  AddNodePosition(pos);
   FlushChildrenNodePositions();
   dc.TextOut(pos, “True”);
}


void    Tassignment<TViewNode>::CalcSize(TMyDC& dc)
{  TSortIdentifier<TViewNode>      *dest   = Getdest();     dest->CalcSize(dc);
   TSortExpr<TViewNode>            *source = Getsource(); source->CalcSize(dc);

   TSize      asize;
   int      width, height;

   dc.GetTextExtent(“ := “, 4, asize);
   width  = dest.cx + asize.cx + source.cx;  //add widths of children and ‘:=‘ decoration
   height = max(max(dest.cy, source.cy), asize.cy); // maximum of heights.

   SetNodeSize(TSize(width, height));
}
void    Tassignment<TViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  TSortIdentifier<TViewNode>      *dest   = Getdest();
   TSortExpr<TViewNode>            *source = Getsource();

   dest->Draw(dc, pos);
   int   x = pos.x + dest->NodeWidth();
   dc.TextOut(x, pos.y, “ := “);
   x += dc.GetTextExtent(“ := “, 4).cx;  // add width of ‘ := ‘
   source->Draw(dc, TPoint(x, pos.y));
}
```

5 Multiple-view structure editor implementation

This chapter describes the C++ implementation of the structure editor with multiple views. To give a quick overview the following illustration shows all classes defined in the structure editor implementation, excluding the ones that describe the signature (see Figure 4.1). Classes that have a parameter in angle brackets are created using a *class template*. A grey box is used to indicate an ObjectWindows Library class not defined in the structure editor itself. In this illustration, *N* indicates the number of view types defined.

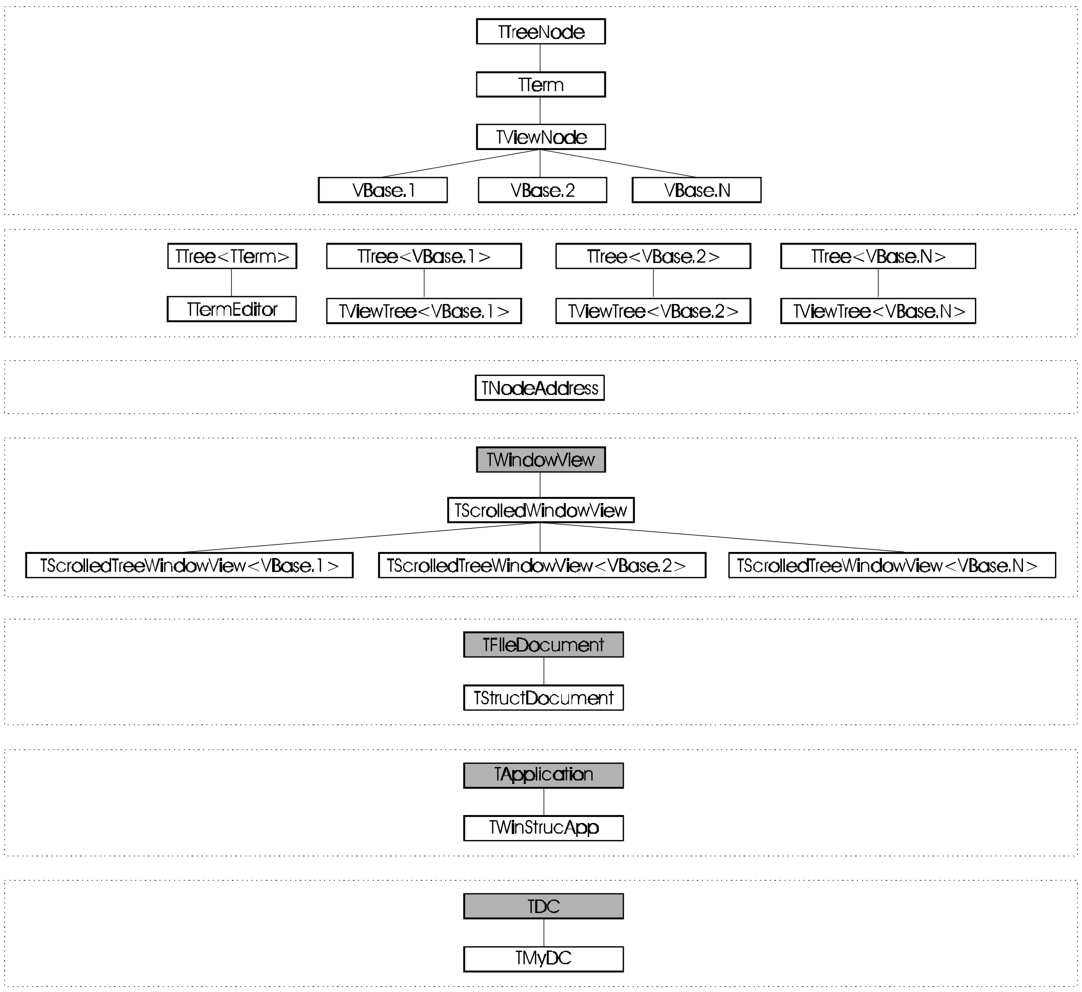


Figure 5.1   Windows structure editor classes

This chapter shows how the Doc / View model is used in practice. Sections 5.1 and 5.2 define the classes used for the Document side in this model. The Document side has no knowledge of the number of view types and view windows currently associated with a document: it simply receives document change requests from views, and broadcasts all actual changes made to the associated views. In addition, the document side is responsible for loading/saving of structures and Windows clipboard support.

The view side is discussed in Section 5.3, where the class TScrolledTreeWindowView is introduced that defines a basic View user interface. This user interface offers a view window with scroll bars, support for a separate focus in each view and a pop-up menu with focus, clipboard and editing menu items. There is no obligation to use this ‘greatest common divisor’ class; when a view type is incompatible with the ideas behind TScrolledTreeWindowView, an implementer can choose to create a view type with different characteristics, independent of TScrolledTreeWindowView.

5.1 Editor class

The class TTermEditor is used to store and manipulate a term tree over a signature with list operators **S**. **S** must be defined in C++ in the way explained in Section 4.1. The purpose of this class is to provide a general term editor class which is completely independent of OWL and the Doc/View model. TTermEditor is included in TStructDocument, which provides Doc/View communications and serves as an interface from the editor object to the clipboard and file system.

The class TTermEditor ensures that its tree of TTerm objects *t* always forms a valid term tree of sort TERMSORT ($t \in Tree_{\Sigma_{TERMSORT}}$). TERMSORT is a constant of type Sort that has been defined along with the signature definition, see Section 4.1.3.

TTermEditor builds on `TTree<TTerm>`, which stores the term tree and provides basic tree operations. It adds the following:

- Operations for term editing as introduced in Section 2.9,
- Stream input/output operations for the whole term tree or a subtree.

This is the C++ declaration of class TTermEditor:

```
class TTermEditor: public TTree<TTerm>
{
public:
                    TTermEditor();

   void             Refine(TNodeAddress& address, Operator op);
   void             Delete(TNodeAddress& address);
   void             ListInsert(TNodeAddress& address, int childnumber);
   void             ListDelete(TNodeAddress& address, int childnumber);

   BOOL             ReadSubtreeFromStream(istream& is, TNodeAddress& addr);
   BOOL             ReadFromStream(istream& is);
   inline void      WriteSubtreeToStream(ostream& os, TNodeAddress& addr);
   inline void      WriteToStream(ostream& os);

private:
   BOOL             ReadSubtreeFromStream(istream& is, TTerm *parent, int cn);
   void             InsertHole(TTerm *parent, int cn);
   void             InsertOperator(TTerm *parent, int cn, Operator op);
};
```

The constructor has no parameters and creates a term tree consisting of a single hole of sort TERMSORT. There is no destructor because the `TTree<TTerm>` base class destructor is sufficient.

In the following descriptions of the following public member functions of class TTermEditor, the current term in the editor is referred to as *t*.

```
void            Refine(TNodeAddress& address, Operator op);
```

Implements the *refine* operator defined in Section 2.9.3 to the hole indicated by `address`. The only difference is that, when `op` refers to a list operator $g{:}\{f_1{:}s_1^*\} \rightarrow s_0$, a single hole of sort s_1 is added as a child to the new list operator node.

t := refine(t, address, op)

and in addition, if `op` is a list operator:

t := insertchild(t, address, 0, Δ_{s_1} *)*

```
void              Delete(TNodeAddress& address);
```

Replaces the subtree at `address` by a hole of the same sort.

t := replace(t, address, Δ_s*),* where s is the sort of the subtree pointed to by `address`.

```
void              ListInsert(TNodeAddress& address, int childnumber);
```

This member function is similar to the *insertchild* operation defined in Section 2.9.4. Assume that address refers to a list operator $g:\{f_1:s_1^*\}\rightarrow s_0$. Instead of inserting an arbitrary subtree however, it inserts a hole of sort s_1 at position `childnumber`.

t := insertchild(t, address, childnumber, Δ_{s_1} *)*

```
void              ListDelete(TNodeAddress& address, int childnumber);
```

This member function implements the *deletechild* operation defined in Section 2.9.5. Of course, t(`address`) must be a list operator with at least `childnumber+1` children.

t := deletechild(t, address, childnumber)

The next member functions are related to stream I/O. They can read from or write to a stream a subtree or the entire term tree. They will be used later for implementing the File Open and Save as well as the clipboard Cut, Copy and Paste functionality.

The subtree is written to a stream as text, which means that it can be viewed/edited using any ASCII text editor. A term is formatted according to the definition of $Term_\Sigma$ in Section 2.3: operator name, opening square bracket, children separated by commas, closing square bracket. If a node has no children, no empty square bracket pair will be generated. And if a node has data associated with it, this data will be written directly after the operator name, enclosed in curly brackets.

```
BOOL              ReadSubtreeFromStream(istream& is, TNodeAddress& addr);
```

Replaces the subtree `addr` by a subtree read from input stream `is`.

t := replace(t, addr, <tree from stream is>)

```
BOOL              ReadFromStream(istream& is);
```

Replaces entire term tree by tree from input stream `is`.

t := <tree from stream is>

```
inline void       WriteSubtreeToStream(ostream& os, TNodeAddress& addr);
```

Writes a subtree to the output stream `os` using the << operator defined with TTerm. In fact, this is the entire function body: `os << NodeAt(addr)`.

```
inline void       WriteToStream(ostream& os);
```

Writes the entire term tree *t* to the output stream `os` using the << operator defined with TTerm. This function does: `os << GetRoot()`.

To create an operator object given an `OP_g` constant, the editor uses the template function `CreateOperatorNode`. Its first parameter is a dummy which allows the compiler to define a version of this function based on the desired base class.

```
template<class T>
T             *CreateOperatorNode(T *,const Operator op)
{
   T            *node = 0;

   switch(op) {
   case OP_g:     node = new Tg<T>; break;
   case OP_h:     node = new Th<T>; break;
   ...
   // and so on for every operator in the signature
   }
   return node;
}
```

Example: to create an object for an operator **g**, having TTerm as base class, the function must be invoked in this way:

```
TTerm       *op_g_node = CreateOperatorNode((TTerm *)0, OP_g);
```

5.2 Document class

The structure editor is a Doc/View application with a single document class. This class encapsulates a TTermEditor object and provides the framework it needs to interface to the views according to the Doc/View model. Furthermore, it makes concrete use of TTermEditor's stream I/O member functions in the implementation of clipboard support and file I/O.

The picture below shows the relationships between TStructDocument, the file system and clipboard and the views associated to this document.

A view communicates to the document by calling one of its member functions. A document communicates to its views by using `NotifyViews` to broadcast view notifications.

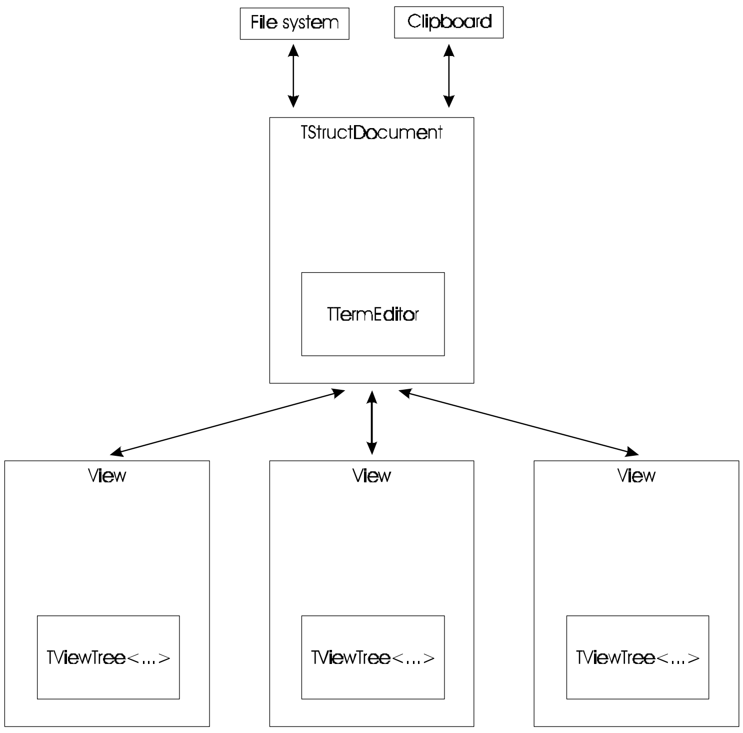


Figure 5.2 Relationships between structure editor components

Here is the C++ declaration for the TStructDocument class:

```
class     TStructDocument : public TFileDocument
{
public:
    TStructDocument(TDocument* parent = 0);

   // editing functions
   void              Refine(TNodeAddress& address, Operator op, TWindow *window = 0);
   void              Delete(TNodeAddress& address);
   void              ListInsert(TNodeAddress& address, int childnumber);
   void              ListDelete(TNodeAddress& address, int childnumber);
   void              GlobalFocus(TNodeAddress& address);

   TTree<TTerm>     *GetTree() {return &Editor;}

   // support for Active Context View
   void              GetActiveAddress(TNodeAddress& address) const;
   void             SetActiveAddress(TNodeAddress& address);

   // clipboard support
   void             Cut(TNodeAddress& address, HWND);
   void             Copy(TNodeAddress& address, HWND);
   void             Paste(TNodeAddress& address, HWND);
   BOOL             ClipboardOK(const Sort sort, HWND) const;



    // file I/O: implement virtual methods of TDocument
   BOOL             Open(int mode, const char far* path=0);
   BOOL             Commit(BOOL force = FALSE);
   BOOL             Revert(BOOL clear = FALSE);

private:
   TTermEditor      Editor;
   TTerm           *ActiveNode;
   static UINT      ClipboardFormat;
};
```

The constructor has no parameters and sets `ActiveNode` to the root of the term tree.

There is no need for a destructor because the `Editor` data member will be destroyed automatically.

Here are the descriptions of TStructDocument's public member functions:

```
void             Refine(TNodeAddress& address, Operator op, TWindow *window = 0);
void             Delete(TNodeAddress& address);
void             ListInsert(TNodeAddress& address, int childnumber);
void             ListDelete(TNodeAddress& address, int childnumber);
```

A view calls one of these member functions when it wants to perform the corresponding edit operation. TStructDocument, in turn, calls `Editor`'s corresponding member function. After this, it sets the dirty flag to indicate that the document has been changed (and will need to be saved) and calls `NotifyViews(vnMirrorSubtree, address)` to notify all associated views of the change.

`Refine` also calls `AskData`, which is a signature-dependent function that can ask for additional user input, depending on which operator is added to the term tree. A typical example is an Identifier operator for which `AskData` would ask, using a string input dialog box, the identifier name. It is `AskData`'s responsibility to attach the additional data to the new node by using TTerm's `SetData` function (described in Section 4.2.2).

```
void             GlobalFocus(TNodeAddress& address);
```

A view calls this function if it wants to set the focus for all views to the same node. TStructDocument passes this request to all associated views by calling `NotifyViews(vnGlobalFocus, address)`. Each view will receive this message and set its focus to the node indicated by `address`.

```
TTree<TTerm>     *GetTree() {return &Editor;}
```

Provides a way for views to access the term tree. In a view constructor, access to the term tree is needed to build the view tree. And when a view gets a `vnMirrorSubtree` message, it uses this function to reach the term subtree it needs to mirror.

```
void             GetActiveAddress(TNodeAddress& address) const;
```

Allows an Active Context-type view to get the address of the active node. This function is also used to set the initial focus when a view is created.

```
void             SetActiveAddress(TNodeAddress& address);
```

Set the `ActiveNode` member to the node indicated by `address`. Called by a view when:

- its local focus changes
- its window receives the input focus

```
void             Copy(TNodeAddress& address, HWND);
```

This operation copies the subtree indicated by `address` to the Windows clipboard. If the structure editor clipboard format has not been registered yet (the static `TStructDocument::ClipboardFormat` is still 0), a clipboard format is registered under the name ‘WinStruc Term Sort’

```
if(!ClipboardFormat) {
   ClipboardFormat = cb.RegisterClipboardFormat("WinStruc Term Sort");
}
```

TTerm’s `TermLength` member function is used to find the length of the textual representation of the subtree. A memory block of this length (+1, to accommodate the terminating 0) is allocated and a string stream (`ostrstream`, for writing to memory as opposed to writing to a file) is opened. Using TTermEditor’s `WriteSubtreeToStream` function, the subtree is written to the memory stream:

```
ostrstream  os(p, memlen);
Editor.WriteSubtreeToStream(os, address);
os << ‘\0’;      // write terminating 0
```

Next, the data is stored on the clipboard using the standard `CF_TEXT` format. This means that any application that can paste plain text from the clipboard can paste this subtree representation. The ‘WinStruc Term Sort’ clipboard format registered earlier, is used to store an additional value of type `Sort`, which indicates the sort of the term on the clipboard.

`void             Cut(TNodeAddress& address, HWND);`

This operation removes the subtree consisting of the current node and all its descendants from the document and places it on the Windows clipboard. A hole of the appropriate sort replaces the subtree in the document. It simply calls `Copy`, then replaces the subtree by a hole.

`void             Paste(TNodeAddress& address, HWND);`

If there is a subtree on the Windows clipboard and `address` refers to a node of the same sort, Paste will replace the node by the clipboard subtree.

Paste retrieves the clipboard memory block that holds the subtree and opens an input string stream (for reading from memory). The subtree is created using TTermEditor’s `ReadSubtreeFromStream` member function:

```
istrstream  is(p);
Editor.ReadSubtreeFromStream(is, address);
```

After this, all views are notified of the paste by a call to

`NotifyViews(vnMirrorSubtree, address)`.

`BOOL             ClipboardOK(const Sort sort, HWND) const;`

This function returns TRUE when the clipboard currently holds a subtree of sort `sort`. It can be used by a view to check whether a Paste menu item should be made available to the user or not.

`BOOL             Open(int mode, const char far* path=0);`

This function reads a term tree from an input file (using `TTermEditor ::ReadFromStream`). It returns TRUE on success, FALSE on failure.

`BOOL             Commit(BOOL force = FALSE);`

If the dirty flag is set or `force` is TRUE, this function saves the term tree to the file using `TTermEditor.WriteToStream`.

`BOOL             Revert(BOOL clear = FALSE);`

This function undoes changes to the term tree by restoring the term tree to the last version saved.

5.3 View classes

In the Doc/View model, each view class implements a different type of view which has its own way to present data on the screen and let users manipulate it. A view class is based on the OWL class TView, which provides the interface to a document class and other basic view functionality. A very useful OWL class to base view classes on is TWindowView. This adds windowing capabilities to TView, in order to let a view show its data in its own separate window.

I will now present the general view class TTreeScrolledWindowView, that interfaces to the document class TStructDocument. This general view class offers a user interface based on a view window with scroll bars, support for a per-view independent focus and a pop-up menu with focus, clipboard and editing menu items. Given this class, one can easily create specialized view types, as will be shown in the implementation of an example structure editor in Chapter 6.

5.3.1 TScrolledWindowView

Before describing TTreeScrolledWindowView, I will first describe its base class TScrolledWindowView. This class adds scrolling and automatic scroll bar support to TWindowView. It is an abstract class: to actually use it, you need to derive a class from it, which implements the pure virtual `CalcSize` member function.

```
class TScrolledWindowView : public TWindowView
{
public:
   TScrolledWindowView(TDocument& doc, TWindow *parent = 0);

   virtual TSize     CalcSize() = 0;
protected:
    // Message response functions
   void    EvSize(UINT sizeType, TSize&);
   void    AdjustScroller();

   DECLARE_RESPONSE_TABLE(TScrolledWindowView);
};

DEFINE_RESPONSE_TABLE1(TScrolledWindowView, TWindowView)
   EV_WM_SIZE,
END_RESPONSE_TABLE;
```

The constructor passes its parameters to the TWindowView constructor, sets the `WS_VSCROLL` and `WS_HSCROLL` window style flags and attachs a scroller (TScroller) to the window.

This class needs no destructor.

Here are the descriptions of TScrolledWindowView's public and protected member functions:

```
virtual TSize     CalcSize() = 0;
```

This function, supplied by a derived class, must return the size (in pixels) of the current window contents.

```
void             AdjustScroller();
```

This function adjusts the scroller given the result of `CalcSize` and the current size of the window's client area. Only if the window contents are wider than the window itself, a horizontal scroll bar appears. And if the contents are higher, a vertical one appears. When the size of the window contents changes, a derived class must call this function in order to have the scroll bars reflect the new situation.

```
void              EvSize(UINT sizeType, TSize&);
```

This function gets called by OWL when the user changes the window size. In turn, it calls `AdjustScroller`.

5.3.2 TMyDC

TDC is the OWL class that encapsulates Windows GDI device contexts. In the structure editor, it is used for drawing graphics in a window. It has a multitude of member functions to draw points, lines, rectangles, ellipses and many other graphics calls. To customize TDC and add more specialized drawing calls, the class TMyDC is derived from it. This class will be used to draw graphic representations of the term tree in view windows.

The class TMyDC for which the C++ declaration is given below is only an example. It has some functions to support the graphics of the view types defined for the PL4 editor example presented in Chapter 6.

```
class TMyDC: public TDC {
public:
   void          RectangleOutline(TRect& rect);

   void          ShadowRectangle(TRect& rect, TBrush& fill, TPen& olpen, TBrush& shadow,
                                TSize& shadowbottleft = TSize(1, BShadH),
                                TSize& shadowtopright = TSize(RShadW, 2));
   enum Direction {UP, DOWN, LEFT, RIGHT};
   void          Arrow(TPoint& pos, enum Direction);
   void          Plus(TPoint& pos);
};
```

The example TMyDC has the following member functions.

```
void            RectangleOutline(TRect& rect);
```

Draws only the outline of a rectangle, using the current Pen.

```
void            ShadowRectangle(TRect& rect, TBrush& fill, TPen& olpen, TBrush& shadow,
                               TSize& shadowbottleft = TSize(1, BShadH),
                               TSize& shadowtopright = TSize(RShadW, 2));
```

Draws a rectangle with a shadow towards the bottom right.

```
void            Arrow(TPoint& pos, enum Direction);
```

Draws an triangle of a predefined size and color (defined in `mydc.h`) pointing UP, DOWN, LEFT or RIGHT with its tip is at `pos`.

```
void            Plus(TPoint& pos);
```

Draws a plus of predefined size and color, centered around `pos`.

View user interface implementers can include any drawing functionality they want in their own TMyDC.

5.3.3 TScrolledTreeWindowView

The class TScrolledTreeWindowView offers a basic user interface for editing and displaying a term tree in a scrolled window view.

A view window has an independent *focus*, which indicates the currently active node in the term tree. The focus can be on a different node in different view windows, to make it possible to use multiple views to look at different sections of a single document and, for instance, use Copy and Paste to copy parts of a document from one section to another.

TScrolledTreeWindowView uses a *pop-up menu* to let the user select commands that apply to the focus node (and possibly all its descendants). This popup menu appears inside the view window client area (as opposed to normal menus, which drop down from the menu bar) when the user presses the ENTER key or the right mouse button. The following illustration shows a popup menu from the PL4 example editor, described in Chapter 6:

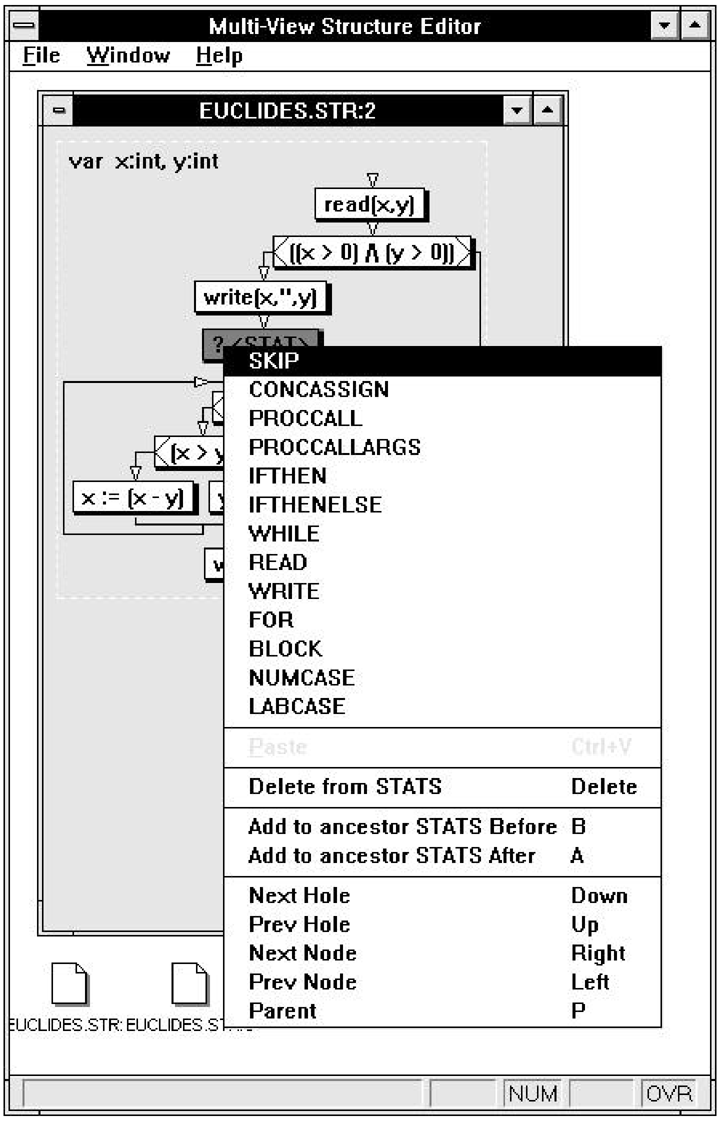


Figure 5.3 Popup menu when focus is on a PL4 STAT hole

Before the popup menu comes up, the focus moves to the node indicated by the coordinates of the mouse click (determined using `TViewTree::WhichNode(x,y)`). Which items are shown in the popup menu depends mainly on the focus node. The meaning of all popup menu commands, and their availability, will be described now.

- **Fold/Unfold**
  If this view's focus is on a node with descendants, 'Fold' will remove it and its descendants from the View display. A marker will be placed in the tree display to indicate that this node, and all its descendants, is hidden. 'Show descendants' will cause a node's hidden descendants to reappear on the screen.

  Keyboard equivalent: 'F'.

  Mouse equivalent: click left mouse button on a node that already has the focus to toggle it between folded and unfolded.

- **Refine**
  If this view's focus is on a hole of sort *s*, the top of the popup menu will list all operators in the signature that match the hole's sort ($\mathbf{G}_s$). By choosing one of these items, the structure editor will add the corresponding operator to the term tree. In addition, for an aggregate

operator, the editor automatically adds holes corresponding to each of the operator's arguments. For a list operator, a single hole is added.

- **Cut & Copy**
  The clipboard Cut and Copy commands are available if the focus is not on a hole. The view calls the corresponding TStructDocument member function to perform the clipboard operation.

  Keyboard equivalent to Cut: Ctrl+'X'.
  Keyboard equivalent to Copy: Ctrl+'C'.

- **Paste**
  If this view's focus is on a hole, 'Paste' will be on the popup menu. However, it will be greyed unless there is a subtree on the Windows clipboard and the clipboard sort equals the hole sort (checked by using `TStructDocument::ClipboardOK`).

  Keyboard equivalent: Ctrl+'V'.

- **Delete**
  If the focus is not on a hole, this command will be available to replace the focus node and all its descendants by a hole.

  Keyboard equivalent: Delete.

- **Add Hole to g**
  This menu item is shown only if the focus is on a list operator **g**. When selected, the editor will add a hole to **g**, after its existing children.

  Keyboard equivalent: 'H'.

- **Add Hole to ancestor g Before/After**
  This menu item is shown only if there is a list operator in the path from focus to root. Assume **g** is the closest list operator ancestor of the focus node. The purpose of this feature is to insert a hole to **g**, either before or after **g**'s child node on the root path.

  A diagram that depicts the root path (from focus node through root node), and the closest list operator ancestor **g** on it, will clarify this operation.

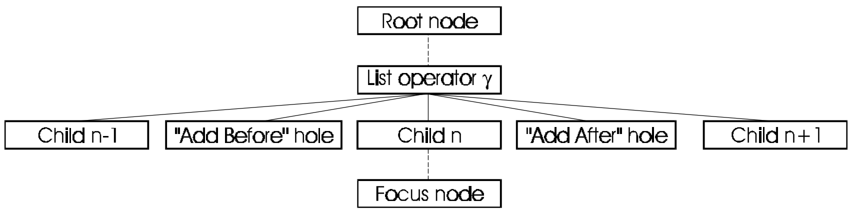


Figure 5.4 Result of Add before/after

  Keyboard equivalent to Add Before: 'B'.
  Keyboard equivalent to Add After: 'A'.

- **Delete from g**
  This item is available if this view's focus is on a hole, child of a list operator **g**. When the user selects the item, the structure editor removes the focused list element from **g**.

  Keyboard equivalent: Delete.

In addition to the popup menu, the following operations exist to change the focus:

- **Focus change**
  Click left mouse button on a node in the view tree to change the focus only in the active view.

- **Global focus change**
  Clicking the left mouse button while pressing SHIFT changes the focus in all views of the active document.

The class TScrolledTreeWindowView is a template class; it takes the view node base class as a parameter. As shown Figure 5.1, this yields a view class TScrolledTreeWindowView<$\mathbf{b}_i$> for every view node base class $\mathbf{b}_i$.

```
template<class T>
class TScrolledTreeWindowView: public TScrolledWindowView
{
public:
    TScrolledTreeWindowView(TStructDocument& doc, TWindow *parent = 0);
    static const char far* StaticName();

    //
    // overridden virtuals from TView
    LPCSTR   GetViewName(){ return StaticName();}

    TSize    CalcSize();

protected:
    virtual  char    GetOpKey( Operator op) { return GetDefaultOpKey(op); }
    virtual  char   *GetOpMenuString( Operator op) { return GetDefaultOpMenuString(op); }

    void      Paint(TDC&, BOOL erase, TRect&);
    LRESULT   EvCommand(UINT id, HWND hWndCtl, UINT notifyCode);

    // Message response functions
    BOOL      VnMirrorSubtree(TNodeAddress *address);
    BOOL      VnGlobalFocus(TNodeAddress *address);
    BOOL      VnNewActive(TNodeAddress *address);

    void      EvSetFocus(HWND);
    void      EvLButtonDown(UINT ModKeys, TPoint& point);
    void      EvRButtonDown(UINT ModKeys, TPoint& point);
    void      EvLButtonDblClk(UINT ModKeys, TPoint& point);
    void      EvChar(UINT key, UINT repeatcount, UINT flags);
    void      EvKeyDown(UINT key, UINT repeatcount, UINT flags);
    void      EvKeyUp(UINT key, UINT repeatcount, UINT flags);

    void      ShowMenu(const TPoint&);
    void      HandleButtonDown(TPoint& point, BOOL nofold = FALSE);
    void      HandleLButtonDown(TPoint& point);
    void      ToggleFold();

    void      ChangeFocus(T *);
    void      Parent();
    void      PrevNode();
    void      NextNode();
    void      PrevHole();
    void      NextHole();

    void      AddHoleToList();
    void      AddHoleBeforeAfter(WPARAM);
    void      Delete();
    void      Cut();
    void      Copy();
    void      Paste();

    DECLARE_RESPONSE_TABLE(TScrolledTreeWindowView);

    TViewTree<T>     Tree;
    TStructDocument *StructDoc;  // same as Doc member, but cast to derived class
    BOOL             ShiftDown;
};
```

Most of the member functions are straightforward implementations of the corresponding menu items. The functions that are relevant to understanding the communications between a TStructDocument object and its associated views according to the Doc/View model will be discussed briefly. All view member functions have access to the document object this view belongs to by the `StructDoc` pointer. They can call TStructDocument member functions to request changes to the term tree. In turn, TStructDocument broadcasts a view notification `vnXxx` (using `NotifyViews`) if a change occurs that all associated views should know about.

```
BOOL      VnMirrorSubtree(TNodeAddress *address);
```

This function is called when the document broadcasts the view notification `vnMirrorSubtree`. This indicates that the subtree indicated by `address` of the term tree has been changed, so the view has to update the corresponding subtree in its own view tree. The view calls `TViewTree::CreateMirror` (see Section 4.3.3) to do this.

```
BOOL      VnGlobalFocus(TNodeAddress *address);
```

This function is linked to the view notification message `vnGlobalFocus`, which indicates that all views have to set their focus to the node indicated by `address`. This function will set the focus for this view to the requested node.

```
BOOL      VnNewActive(TNodeAddress *address);
```

This function is linked to the view notification message `vnNewActive`, which is used by an active context view (see Chapter 6) so that it can show the active context of the focus node in the active window. This message is broadcast when a view calls `TStructDocument::SetActiveAddress`, which it does when its focus changes or when its window becomes active.

As said before, operators appear in the view window popup menu when the focus is on a hole. The following function generates the string to be shown in the menu given an operator costant:

```
virtual  char   *GetOpMenuString( Operator op) { return GetOpName(op); }
```

The default behavior is to show the operator name (as produced by `GetOpName`()). If a view types needs to override the standard menu strings, the view class should be a derived class of TScrolledTreeWindowView that overrides `GetOpMenuString`.

The function `GetOpKey` can be used to provide keyboard shortcuts to operators.

```
virtual  char    GetOpKey( Operator op) { return 0; }
```

The default behavior is to return 0 (no keyboard shortcut for any operator). In a derived class, a view can implement a real `GetOpKey` function that assigns keyboard shortcuts to operators.

6 Example: Multiple-view PL4 editor

This chapter describes an example editor based on the language PL4 from the Compilers 1 lecture notes ([4], Section 3.11, page 47). It has three view types:

- Listing view
- Flow diagram view
- Active context view

The first section will describe the implementation of the PL4 grammar as a signature in C++. After this, the implementation of the view types is discussed.

6.1 PL4 signature

The language PL4 is a programming language with block structure (both for statements and expressions), procedures and functions. All variables have one of the types *int*, *bool*, *char* or *real*. It has the following elementary statements:

- **skip**
- **concurrent assignment**
- **selection statements:**
  - if-then
  - if-then-else
- **repetition statements:**
  - while
  - repeat
  - for
- **case statements:**
  - numcase (where the alternatives are implicitly numbered 1,2,...),
  - labcase (all alternatives labeled with integer or character constants).
- **procedure call statements:**
  - procedure call with no parameters
  - procedure call with parameters
- **I/O statements:**
  - read
  - write

Section 3.11 of [4] lists the grammar G_4 of PL4. [4], Section 3.10, page 45, contains the following PL4 example program, which reads two numbers and computes their greatest common divisor:

```
|[  var x:int, y:int
|   read(x,y)
   ;if x>0 /\ y>0 then
       write(x,’ ‘,y);
      ;while x /= y do
          if x > y then
             x := x-y
          else
             y := y-x
          fi
       od
      ;write(‘ ‘, x)
    fi
]|
```

From the PL4 grammar, a signature with list operators was derived, and implemented in C++ using the pattern given in Section 4.1.3. All signature-dependent definitions and declarations are isolated in a single module, "SIGN_PL4". It consists of a header file "SIGN_PL4.h" and a source file "SIGN_PL4.h", both of which are listed in Appendix B.

The header file "SIGN_PL4.h" contains the sort and operator template classes, which were indirectly defined by using the DEFINE_OPxxx macros. In addition, it contains a function template for CreateOperatorNode. The source file "SIGN_PL4.cpp" contains the functions AskData, GetOpSort, GetSortName and GetOpName.

The start symbol of G_4 is PROG. This is reflected in the C++ signature declaration by:

```
const Sort TERMSORT = SORT_PROG;
```

The last part of file "SIGN_PL4.h" is the definition of a function template for CreateOperatorNode (introduced in Section 5.1) which creates an operator object, given the corresponding operator constant.

I will now describe the functions that you can find in appendix B, file "SIGN_PL4.cpp". As described in Section 5.2, TStructDocument::Refine calls AskData, which is a function to be supplied by the signature implementer. Depending on the operator added by the refine operation, this function asks for additional data. For the PL4 editor, these are the operators that need additional data and the type of data needed:

- NUMVALUE, NUMLABEL: a numeric constant
- CHARVALUE, CHARLABEL: a character constant
- ID: a string (the identifier name)

This is the AskData function for the PL4 example signature:

```
void      AskData(TWindow *window, TTerm *node)
{
   char      s[MAXOPDATALEN+1], v[20];
   char      h[MAXOPNAMELEN+51];

   s[0] = 0;
   wsprintf(h, "Operator %s additional info", node->GetOpName());
   switch(node->GetOperator()) {
   case OP_NUMVALUE:
   case OP_NUMLABEL:
      TInputDialog(window, h, "Enter numeric value:",
                   s, sizeof s).Execute();
      ostrstream(v, sizeof v)<<atof(s)<<'\0';
      node->SetData(v, strlen(v)+1);
      break;
   case OP_CHARVALUE:
   case OP_CHARLABEL:
      TInputDialog(window, h, "Enter a character:",
                   s, sizeof s).Execute();
      s[1] = s[0];
      s[0] = '\'';
      s[2] = '\'';
      s[3] = 0;
      node->SetData(s, 4);
      break;
   case OP_ID:
      TInputDialog(window, h, "Enter an identifier name",
                   s, sizeof s).Execute();
      node->SetData(s, strlen(s)+1);
      break;
   }
}
```

The OWL class TInputDialog shows a dialog box on the screen and asks the user for a string. String *h* is displayed in the dialog box title bar.

The function GetOpSort returns the result sort of operator op.

GetSortName returns the name of a given sort by using the sort constant as an index in an array of sort names.

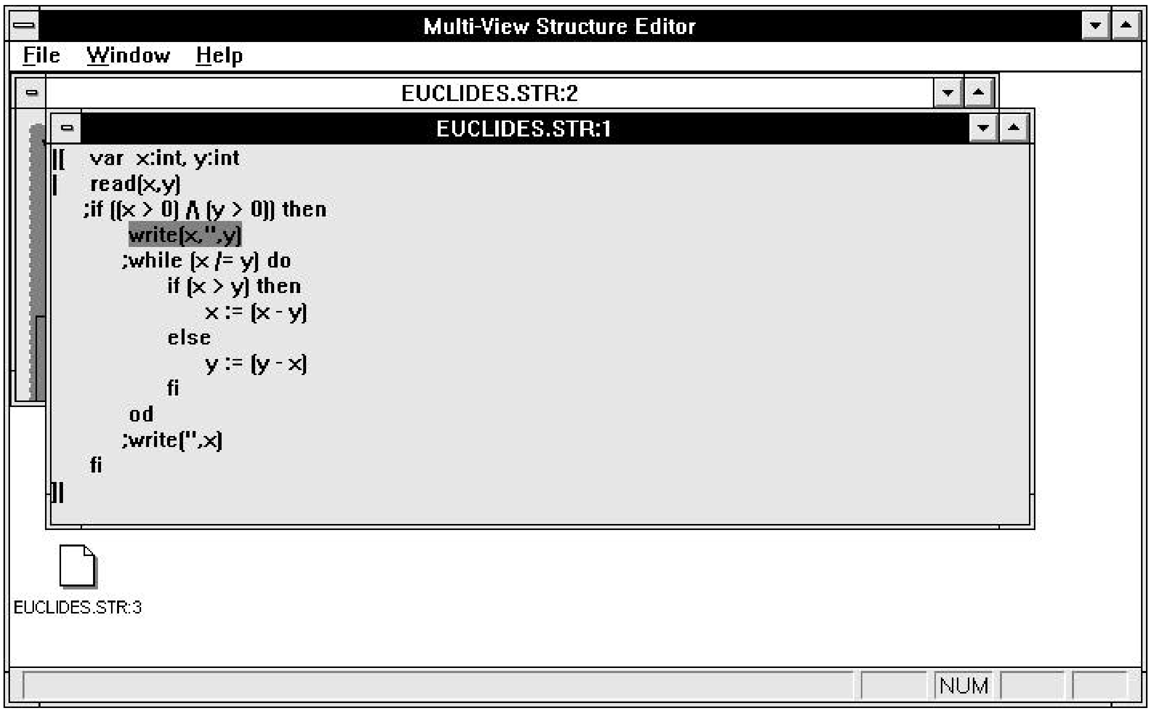
`GetOpName` returns the name of a given operator. First it checks whether the operator constant is less than `OP_FIRST`, which would indicate that the operator is a hole. If this is the case, by definition the operator constant is equal to the corresponding sort constant, and `GetOpName` returns the concatenation of sort name and "Hole" as the operator name. If the operator is not a hole, its name is retrieved from an array of operator names:

```
char   *GetOpName(Operator op)
{
   if(!op)
      return 0;
   if(op<OP_FIRST) {
      static char   nbuf[MAXOPNAMELEN+1];
      wsprintf(nbuf, "%sHOLE", GetSortName((Sort)op));
      return nbuf;
   } else {
      return OpName[(int)(op-OP_FIRST)];
   }
}
```

This completes the description of the PL4 signature. The following sections present the implementation of the three view types that exist in the example PL4 editor.

6.2 Listing View

The listing view shows the PL4 term as a listing. The following screen shot is a listing view of the Euclides example program:



```
|[  var  x:int, y:int
|   read(x,y)
   ;if ((x > 0) ∧ (y > 0)) then
        write(x,'',y)
      ;while (x /= y) do
           if (x > y) then
                x := (x - y)
           else
                y := (y - x)
           fi
        od
      ;write('',x)
   fi
]|
```

Figure 6.1 PL4 Listing View

The implementation of the listing view is given in Appendix B, file "`LIST_PL4.cpp`". This discussion shows its main parts. The listing view is based on the basic view user interface template class TScrolledTreeWindowView, defined in Section 5.3.3.

6.2.1 The view base class TListingViewNode

The view base class for nodes in a Listing view tree is TListingViewNode. It is a derived class of TViewNode (as should all view base classes) with no additional data members.

```
class   TListingViewNode : public TViewNode {
public:
                            TListingViewNode(int n = 0, int fixed = FALSE);
   inline TListingViewNode *GetChild(int n) const;
```

```
    void                    CalcSizeFolded(TMyDC&);
    void                    DrawFolded(TMyDC&, TPoint& pos);
    void                    CalcSizeString(TMyDC&, const char const *);
    void                    DrawString(TMyDC&, TPoint&, const char const *);
    void                    CalcSizeHole(TMyDC&);
    void                    DrawHole(TMyDC&, TPoint&);
    void                    DoPosFocusFolded(TMyDC& dc, TPoint& pos);
    void                    CalcSizeDelimiters(TMyDC& dc, \
                                          char *start, char *between, char *end);

    void                    DrawDelimiters(TMyDC& dc, TPoint& pos, \
                                          char *start, char *between, char *end);
    static TViewTree<TListingViewNode>     *ListingTree;
};
```

The member functions exist to avoid code duplication: calculations or drawing operations that need to be performed by many `CalcSize` or `Draw` functions are implemented as member functions of TListingViewNode.

`void                CalcSizeFolded(TMyDC&);`

Computes the size of this node when it is folded. Uses `SetNodeSize` to store this information in the TViewNode base class.

`void                DrawFolded(TMyDC&, TPoint& pos);`

Draws this node folded, at position `pos`. In the listing view, a folded node is displayed as:

`+ OPERATOR NAME <RESULT SORT NAME>`

`void                 CalcSizeString(TMyDC&, const char const *);`

Computes the size of the bounding box around a given text string . Uses `SetNodeSize` to store this information in the TViewNode base class. An example of this is the Identifier operator that retrieves the associated data (the identifier name) and calls `CalcSizeString` to have the bounding box size calculated.

```
   void  TID<TListingViewNode>::CalcSize(TMyDC& dc)
   {   CalcSizeString(dc, (char *)GetData()); }
```

`void                 DrawString(TMyDC&, TPoint& pos, const char const *);`

Draws a string to the given dc at position `pos`.

```
   void  TID<TListingViewNode>::Draw(TMyDC& dc, TPoint& pos)
   {   DrawString(dc, pos, (char *)GetData()); }
```

`void                 CalcSizeHole(TMyDC&);`

Should be invoked when this node is a hole. Computes the size of the bounding box around the hole graphical representation for this node. Uses `SetNodeSize` to store this information in the TViewNode base class. Example usage:

```
   void  TEXPRHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
   {   CalcSizeHole(dc); }
```

`void                 DrawHole(TMyDC&, TPoint& pos);`

Draws a hole at position `pos`. In the listing view, a hole looks like this:

`? <OPERATOR NAME>`

Example usage:

```
   void  TEXPRHOLE<TListingViewNode>::Draw(TMyDC& dc, TPoint& pos)
   {   DrawHole(dc, pos); }
```

`void                 DoPosFocusFolded(TMyDC& dc, TPoint& pos);`

Does the following chores:

- AddNodePosition(pos);
- Draw dark grey rectangle if the focus is currently on this node.
- If node is folded, call DrawFolded.

This function is called from within all Draw functions (either directly or indirectly).

```
void                CalcSizeDelimiters(TMyDC& dc, \
                                      char *start, char *between, char *end);
```

Computes the bounding box size of the horizontal concatenation of all children of this node, separated by the given delimiter strings. This is useful for calculating the bounding box size of operators that are represented by their children left-to-right, separated by delimiters. For instance, the PL4 `read` statement operator uses this function in its `CalcSize` function:

```
void  TREAD<TListingViewNode>::CalcSize(TMyDC& dc)
{   CalcSizeDelimiters(dc, "read(", ",", ")");
}
```

```
void                  DrawDelimiters(TMyDC& dc, TPoint& pos, \
                                     char *start, char *between, char *end);
```

Draws this operator node as the horizontal concatenation of all its children, separated by the given delimiter strings. This is useful for drawing operators that are represented by their children drawn left-to-right, separated by delimiters. For instance, the PL4 `read` statement operator uses this function in its `Draw` function:

```
void  TREAD<TListingViewNode>::Draw(TMyDC& dc, TPoint& pos)
{   DrawDelimiters(dc, pos, "read(", ",", ")");
}
```

6.2.2 PL4 Listing CalcSize and Draw functions

After the definition of TListingViewNode and its member functions in “LIST_PL4.cpp”, CalcSize and Draw functions are defined for each operator **g** in the PL4 operator set:

```
void  Tg<TListingViewNode>::CalcSize(TMyDC& dc)
{   ...
}

void  Tg<TListingViewNode>::Draw(TMyDC& dc, TPoint& pos)
{   ...
}
```

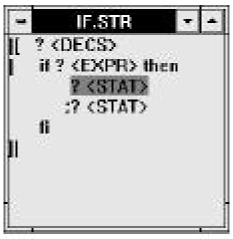
As an example, the `CalcSize` and `Draw` functions of the IFTHEN statement operator are given below:

```
void  TIFTHEN<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TListingViewNode>  *guard = GetGuard(); guard->CalcSize(dc);
      TSortSTATS<TListingViewNode> *stats = GetTrueStatements(); stats->CalcSize(dc);
      TSize    ifsize, thensize, fisize;
      dc.GetTextExtent("if ", 3, ifsize);
      dc.GetTextExtent(" then", 5, thensize);
      dc.GetTextExtent("fi ", 3, fisize);
      SetNodeSize(TSize(max((int)(ifsize.cx+guard->NodeWidth()+thensize.cx),
                              Indent+stats->NodeWidth()),
                         ifsize.cy+stats->NodeHeight()+fisize.cy));
   }
}

void  TIFTHEN<TListingViewNode>::Draw(TMyDC& dc, TPoint& pos)
{   DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSortEXPR<TListingViewNode>  *guard = GetGuard();
      TSortSTATS<TListingViewNode> *stats = GetTrueStatements();
      TSize    ifsize, thensize, fisize;
      dc.GetTextExtent("if ",   3, ifsize);
      dc.GetTextExtent(" then", 5, thensize);
      dc.GetTextExtent("fi ",   3, fisize);
      int x = pos.x+ifsize.cx;
      dc.TextOut(pos, "if ");
      guard->Draw(dc, TPoint(x, pos.y));
      x += guard->NodeWidth();
      dc.TextOut(x, pos.y, " then");
      int y = pos.y+ifsize.cy;
      stats->Draw(dc, TPoint(pos.x+Indent, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "fi");
   }
}
```

For your information: the OWL function `GetTextExtent` calculates the dimensions of a graphical rendering of the given text string (using the current font). `Indent` is a constant that specifies the number of in pixels to move to the right when an indent is needed.

This is what an IFTHEN operator will look like in the Listing View:



```
[  ? <DECS>
   if ? <EXPR> then
        ? <STAT>
      :? <STAT>
   fi
]
```

Figure 6.2 Listing View of IFTHEN operator

6.2.3 TListingView class

The listing view class is created from the TScrolledTreeWindowView template class by supplying the view base class, TListingViewNode:

```
typedef TScrolledTreeWindowView<TListingViewNode> TListingView;
```

TScrolledTreeWindowView contains a response table declaration. However, the response table definition cannot be part of a class template and must be supplied separately for the TListingView class. This actually proves to be an advantage, because a separate response table definition for each view type allows us to specify for each view type which events it should respond to.

The listing view needs to respond to all events that TScrolledWindowView can handle, so it has the full response table.

```
DEFINE_RESPONSE_TABLE1(TListingView, TScrolledWindowView)
   EV_WM_CHAR,
   EV_WM_LBUTTONDOWN,
   EV_WM_RBUTTONDOWN,
   EV_WM_LBUTTONDBLCLK,
   EV_WM_KEYDOWN,
   EV_WM_KEYUP,
   EV_WM_SETFOCUS,
   EV_VN_MIRRORSUBTREE,
   EV_VN_GLOBALFOCUS,

   EV_COMMAND(CM_PREVNODE,       PrevNode),
   EV_COMMAND(CM_NEXTNODE,       NextNode),
   EV_COMMAND(CM_PREVHOLE,       PrevHole),
   EV_COMMAND(CM_NEXTHOLE,       NextHole),
   EV_COMMAND(CM_DELETE,         Delete),
   EV_COMMAND(CM_CUT,            Cut),
   EV_COMMAND(CM_COPY,           Copy),
   EV_COMMAND(CM_PASTE,          Paste),
   EV_COMMAND(CM_PARENT,         Parent),
   EV_COMMAND(CM_FOLD,           ToggleFold),
   EV_COMMAND(CM_ADDHOLETOLIST,  AddHoleToList),
   EV_COMMAND_AND_ID(CM_ADDHOLEBEFORE,  AddHoleBeforeAfter),
   EV_COMMAND_AND_ID(CM_ADDHOLEAFTER,   AddHoleBeforeAfter),
END_RESPONSE_TABLE;
```

The following function overrides the member function `StaticName` that would normally be generated by the TScrolledTreeWindowView template class. It returns a string: the name of the view. This string is used by the document manager to show a menu that lists the view types to choose from when adding a view.

```
const char far* TScrolledTreeWindowView<TListingViewNode>::StaticName()
{
   return "Listing View";
}
```

The OWL macro below defines a template class TListingTemplate by associating a document class with a view class.

```
DEFINE_DOC_TEMPLATE_CLASS(TStructDocument, TListingView, TListingTemplate);
```

Instantiating TListingTemplate is the OWL way of registering a combination of Document class and View class with the Document Manager.

```
TListingTemplate listingTpl("Listing View", "*.STR", 0, "STR",
                            dtAutoDelete | dtUpdateDir |dtAutoOpen);
```

The constructor parameters are: Text description of template, file name filter, default directory (0 indicates current directory), file extension, flags.

Flags values used:

`dtAutoDelete`:	Close and delete the document object when the last view is closed
`dtUpdateDir`:	Update directory with file dialog directory
`dtAutoOpen`:	Open a document upon creation

6.3 Flow Diagram View

The flow diagram view shows the PL4 term as a flow diagram. The following screen shot shows what a flow diagram of the Euclides example program looks like:

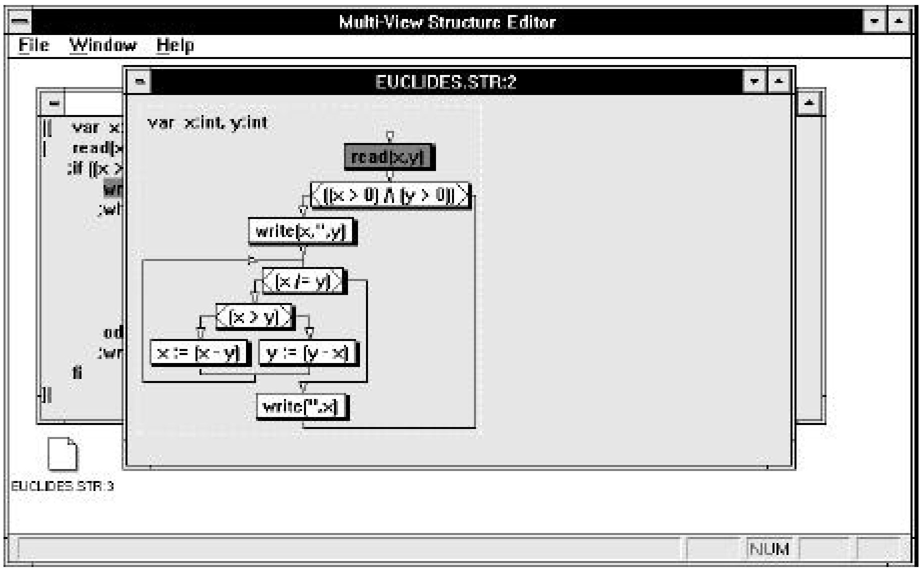


Figure 6.3 PL4 Flow Diagram View

Each statement is rendered inside a rectangle that casts a thin shadow towards the bottom right, to give it a 3D effect. Each PL4 scope (a block, procedure and function) is bounded by a dotted rectangle. In the top left corner of the scope's bounding rectangle, the flow diagram view shows the declarations that belong to a particular scope. The flow diagram of Figure 6.3 has a single scope with two integer variables, x and y.

The implementation of the listing view is given in Appendix B, file "`FLOW_PL4.cpp`". In this discussion its main parts are explained. Just like the listing view, the flow diagram view is based on the basic view user interface template class TScrolledTreeWindowView, defined in Section 5.3.3.

6.3.1 The view base class TFlowViewNode

The view base class for nodes in a Flow Diagram view tree is TFlowViewNode. It is a derived class of TViewNode (as should all view base classes) with one additional data member: an `int ConnX`. This holds the X coordinate, relative to a node's bounding box, where its connecting lines (to the top and bottom of the bounding box) should be drawn:

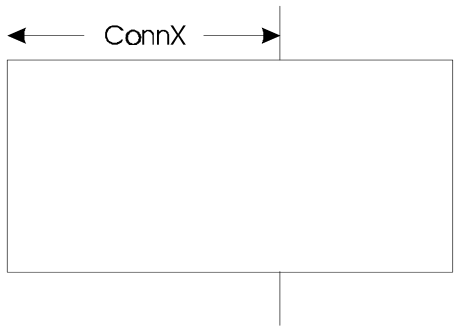


Figure 6.4 Meaning of ConnX

```
class   TFlowViewNode : public TViewNode {
public:
                         TFlowViewNode(int n = 0, int fixed = FALSE);
   inline TFlowViewNode *GetChild(int n) const;
   void                  CalcSizeFolded(TMyDC&);
   void                  DrawFolded(TMyDC&, TPoint& pos);
   void                  CalcSizeString(TMyDC&, const char const *);
   void                  DrawString(TMyDC&, TPoint&, const char const *);
   void                  CalcSizeHole(TMyDC&);
   void                  DrawHole(TMyDC&, TPoint&);
   BOOL                  DoPosFocusFolded(TMyDC& dc, TPoint& pos);
   void                  CalcSizeDelimiters(TMyDC& dc, \
                                  char *start, char *between, char *end);

   void                  DrawDelimiters(TMyDC& dc, TPoint& pos, \
                                  char *start, char *between, char *end);
   void                  DrawStatBox(TMyDC& dc, TPoint& pos);
   inline int            GetConnX() const;
   inline void           SetConnX(int connx);

   static TViewTree<TFlowViewNode>     *FlowTree;

private:
   int                   ConnX;
};
```

The member functions exist to avoid code duplication: calculations or drawing operations that need to be performed by many `CalcSize` or `Draw` functions are implemented as member functions of TFlowViewNode. Many member functions serve the same function as the corresponding ones of TListingViewNode, so their explanation will not be repeated here.

```
void                DrawStatBox(TMyDC&, TPoint& pos);
```

Draws a statement box (with shadow) at position pos. The size will be retrieved from the TViewNode class, function NodeSize(). This function uses `TMyDC::ShadowRectangle` to draw the rectangle.

```
inline int          GetConnX() const;
```

Returns the value of the `ConnX` data member.

```
inline void         SetConnX(int connx);
```

Sets the `ConnX` data member to the given value.

6.3.2 PL4 Flow Diagram CalcSize and Draw functions

After the definition of TFlowViewNode and its member functions in "`FLOW_PL4.cpp`", `CalcSize` and `Draw` functions are defined for each operator **g** in the PL4 operator set:

```
void  Tg<TFlowViewNode>::CalcSize(TMyDC& dc)
{   ...
}
void  Tg<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
```

```
{   ...
}
```

In the flow diagram view, the `CalcSize` function not only computes the resulting bounding box size given a node's children bounding box size. Each `CalcSize` function must also compute the `ConnX` value.

Here are the `CalcSize` and `Draw` functions of the IFTHEN statement operator:

```
void  TIFTHEN<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TFlowViewNode>  *guard = GetGuard(); guard->CalcSize(dc);
      TSortSTATS<TFlowViewNode> *stats = GetTrueStatements(); stats->CalcSize(dc);

      TSize gsize = guard->NodeSize() + TSize(guard->NodeHeight()+TextBorderY*2, TextBorderY*2);

      SetNodeSize(TSize(max(stats->GetConnX() + HConnLen*2 + gsize.cx ,
                        stats->NodeWidth() + DistX),
                     gsize.cy+DistY+stats->NodeHeight()+VConnLen));
      SetConnX(max(NodeWidth()/2, stats->GetConnX()+HConnLen+gsize.cx/2));
   }
}

void  TIFTHEN<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{
   if(DoPosFocusFolded(dc, pos)) {
      TSortEXPR<TFlowViewNode>  *guard = GetGuard();
      TSortSTATS<TFlowViewNode> *stats = GetTrueStatements();
      int                        d, d2, gx, rx;
      TSize                      gsize;

      d2    = guard->NodeHeight()+TextBorderY*2; d = d2/2;
      gsize = guard->NodeSize() + TSize(d2, TextBorderY*2);
      gx    = pos.x + GetConnX() - gsize.cx/2;


      dc.ShadowRectangle(TRect(TPoint(gx, pos.y),gsize), FillBrush, Pen, ShadowBrush);
      dc.MoveTo(gx+d, pos.y); dc.LineTo(gx, pos.y+d); dc.LineTo(gx+d, pos.y+d2);
      rx = gx+gsize.cx-1;
      dc.MoveTo(rx-d, pos.y); dc.LineTo(rx, pos.y+d); dc.LineTo(rx-d, pos.y+d2);

      dc.MoveTo(gx, pos.y+d); dc.LineTo(pos.x+stats->GetConnX(), pos.y+d);
      dc.LineTo(pos.x+stats->GetConnX(), pos.y+gsize.cy+DistY);
      dc.Arrow(TPoint(pos.x+stats->GetConnX(), pos.y+gsize.cy+DistY), TMyDC::DOWN);


      dc.MoveTo(rx, pos.y+d); dc.LineTo(pos.x+NodeWidth()-1, pos.y+d);
      dc.LineTo(pos.x+NodeWidth()-1, pos.y+NodeHeight()-1);
      dc.LineTo(pos.x+stats->GetConnX(), pos.y+NodeHeight()-1);
      dc.LineTo(pos.x+stats->GetConnX(), pos.y+NodeHeight()-VConnLen);
      guard->Draw(dc, TPoint(gx+d, pos.y+TextBorderY));
      stats->Draw(dc, TPoint(pos.x, pos.y+gsize.cy+DistY));
   }
}
```

This is what an IFTHEN operator will look like in the Flow Diagram View:

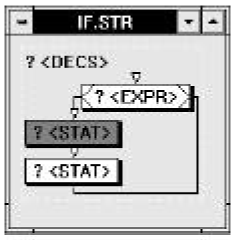


Figure 6.5   Flow Diagram View of IFTHEN operator

6.3.3 TFlowView class

The flow diagram view class is created from the TScrolledTreeWindowView template class by supplying the view base class, TFlowViewNode:

```
typedef TScrolledTreeWindowView<TFlowViewNode> TFlowView;
```

As explained in 6.2.3, the response table definition cannot be part of a class template and must be supplied separately for each view class. Like the Listing View, the flow diagram view needs to respond to all events handled in TScrolledTreeWindowView, so it has the full response table.

```
DEFINE_RESPONSE_TABLE1(TFlowView, TScrolledWindowView)
   EV_WM_CHAR,
   EV_WM_LBUTTONDOWN,
   EV_WM_RBUTTONDOWN,
   EV_WM_LBUTTONDBLCLK,
   EV_WM_KEYDOWN,
   EV_WM_KEYUP,
   EV_WM_SETFOCUS,
   EV_VN_MIRRORSUBTREE,
   EV_VN_GLOBALFOCUS,

   EV_COMMAND(CM_PREVNODE,       PrevNode),
   EV_COMMAND(CM_NEXTNODE,       NextNode),
   EV_COMMAND(CM_PREVHOLE,       PrevHole),
   EV_COMMAND(CM_NEXTHOLE,       NextHole),
   EV_COMMAND(CM_DELETE,         Delete),
   EV_COMMAND(CM_CUT,            Cut),
   EV_COMMAND(CM_COPY,           Copy),
   EV_COMMAND(CM_PASTE,          Paste),
   EV_COMMAND(CM_PARENT,         Parent),
   EV_COMMAND(CM_FOLD,           ToggleFold),
   EV_COMMAND(CM_ADDHOLETOLIST,  AddHoleToList),
   EV_COMMAND_AND_ID(CM_ADDHOLEBEFORE,  AddHoleBeforeAfter),
   EV_COMMAND_AND_ID(CM_ADDHOLEAFTER,   AddHoleBeforeAfter),
END_RESPONSE_TABLE;
```

The following function overrides the member function `StaticName` that would normally be generated by the TScrolledTreeWindowView template class. It returns a string: the name of the view. This string is used by the document manager to show a menu that lists the view types to choose from when adding a view.

```
const char far* TScrolledTreeWindowView<TFlowViewNode>::StaticName()
{
   return "Listing View";
}
```

The OWL macro below defines a template class TFlowTemplate by associating the Document class TStructDocument with the view class TFlowView.

```
DEFINE_DOC_TEMPLATE_CLASS(TStructDocument, TFlowView, TFlowTemplate);
```

The object `flowTpl` is an instance of the class TFlowTemplate. This is the OWL way of registering a combination of Document class and View class with the Document Manager.

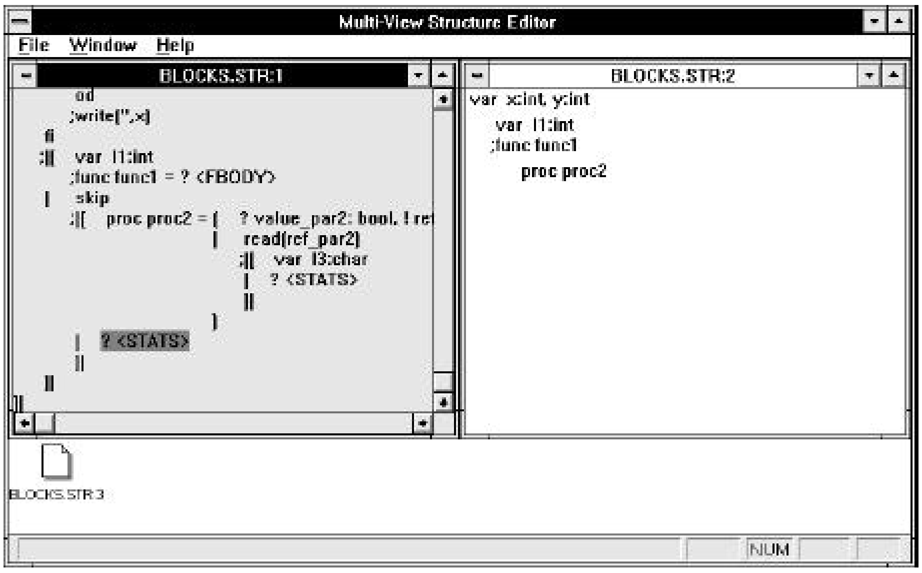
```
TFlowTemplate flowTpl("Flow View", "*.STR", 0, "STR",
                     dtAutoDelete | dtUpdateDir |dtAutoOpen);
```

Here, the text description is “Flow View”. The other parameters are the same as the ones given to the TListingTemplate constructor.

6.4 Active Context View

The active context view depends on another view window. If a view window (listing view or flow diagram view) is active, the Active Context View window shows all variable, procedure and function declarations that are in scope at the location of the focus. The Active Context View window is automatically updated when the focus changes, when declarations are added/removed and when another associated view window becomes active. When none of the associated listing or flow diagram windows are active, the Active Context View window will continue to show the active context for the most recently active view window.

The active focus window has no focus and no popup menu.



```
      od
     ;write('',x)
  fi
 ;||  var I1:int
     ;func func1 = ? <FBODY>
  |   skip
     ;||[   proc proc2 = (   ? value_par2: bool, ! ref
                         |    read(ref_par2)
                             ;||   var I3:char
                              |    ? <STATS>
                              ||
                         )
      |   ? <STATS>
      ||
  ||
||
```

```
var x:int, y:int
   var I1:int
  ;func func1
      proc proc2
```

Figure 6.6 PL4 Active Context View

The context is rendered from the global PROG block (at the top) down to the inner block that contains the focus (at the bottom). Each level of block nesting in the PL4 structure generates an indent in the context view. This done by following the root path from root node (which is always a PROG operator in the PL4 syntax) down to the focus node. Each time an operator is found that has a descendant of sort DECS, PPARS (procedure parameters) or FPARS (function parameters), this descendant is drawn to the screen. The process ends after the focus node has been reached.

The implementation of the listing view is given in Appendix B, file "`CTXT_PL4.cpp`". This discussion covers its main parts. Just like the listing view, the Active Context view is based on the basic view user interface template class TScrolledTreeWindowView, defined in Section 5.3.3. Because the active context view has no focus and no popup menu, there are few events it needs to respond to and its response table is very small.

6.4.1 The view base class TContextViewNode

The view base class for nodes in an Active Context view tree is TContextViewNode. It is a derived class of TViewNode (as should all view base classes) with no additional data members.

```
class   TContextViewNode : public TViewNode {
public:
                         TContextViewNode(int n = 0, int fixed = FALSE);
   inline TContextViewNode *GetChild(int n) const;
   void                  CalcSizeString(TMyDC&, const char const *);
   void                  DrawString(TMyDC&, TPoint&, const char const *);
   void                  CalcSizeHole(TMyDC&);
   void                  DrawHole(TMyDC&, TPoint&);
   void                  CalcSizeDelimiters(TMyDC& dc, \
                                        char *start, char *between, char *end);

   void                  DrawDelimiters(TMyDC& dc, TPoint& pos, \
                                        char *start, char *between, char *end);
   static TNodeAddress      ActiveAddress;
};
```

The member functions exist to avoid code duplication: calculations or drawing operations that need to be performed by many `CalcSize` or `Draw` functions are implemented as member functions of TContextViewNode. All member functions serve the same function as the corresponding ones of TListingViewNode, so their explanation will not be repeated here.

6.4.2 PL4 Active Context View CalcSize and Draw functions

After the definition of TContextViewNode and its member functions in "CTXT_PL4.cpp", CalcSize and Draw functions are defined for each operator **g** in the PL4 operator set:

```
void  Tg<TContextViewNode>::CalcSize(TMyDC& dc)
{   ...
}
void  Tg<TContextViewNode>::Draw(TMyDC& dc, TPoint& pos)
{   ...
}
```

Only the CalcSize and Draw functions of operators that can appear in declarations are implemented:

```
PROG, DECS, TNDECS, TNDEC, TPROCDEC, TFUNCDEC, TID,
INTTYPE, CHARTYPE, REALTYPE, BOOLTYPE,
PPARS, PPARVALUE, PPARREF
FPARS, FPARVALUE
```

All other operators have empty CalcSize and Draw operators:

```
void  TSTATS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TSTATS<TContextViewNode>::Draw(TMyDC&, TPoint&){}

void  TSKIP<TContextViewNode>::CalcSize(TMyDC&) {}
void  TSKIP<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TCONCASSIGN<TContextViewNode>::CalcSize(TMyDC&) {}
void  TCONCASSIGN<TContextViewNode>::Draw(TMyDC&, TPoint&) {}
```

And so on.

The most important CalcSize and Draw functions in the Active Context view are for the PROG operator:

```
void  TPROG<TContextViewNode>::CalcSize(TMyDC& dc)
{
   TSortDECS<TContextViewNode>    *decs  = GetDeclarations(); decs->CalcSize(dc);
   TSize                           sz    = decs->NodeSize();
   TContextViewNode               *node  = this;
   int                             ind   = Indent;

   for(int i = 0; i<ActiveAddress.Length(); i++) {
      node = node->GetChild(ActiveAddress.GetChildNumber(i));
      Sort      sort = node->GetDesiredChildSort(0);
      if(sort==SORT_DECS || sort==SORT_PPARS || sort==SORT_FPARS) {
         TContextViewNode  *child = node->GetChild(0);

         child->CalcSize(dc);
         if(child->NodeHeight()) {
            sz.cy += child->NodeHeight()+DistY;
            sz.cx  = max(sz.cx,ind+child->NodeWidth());
            ind   += Indent;
         }
   }
   SetNodeSize(sz);
}


void  TPROG<TContextViewNode>::Draw(TMyDC& dc, TPoint& pos)
{
   TSortDECS<TContextViewNode>    *decs  = GetDeclarations();
   TPoint                         p = pos;
   TContextViewNode               *node  = this;

   decs->Draw(dc, p); p += TSize(Indent, DistY+decs->NodeHeight());
   for(int i = 0; i<ActiveAddress.Length(); i++) {
      node = node->GetChild(ActiveAddress.GetChildNumber(i));
      Sort      sort = node->GetDesiredChildSort(0);
      if(sort==SORT_DECS || sort==SORT_PPARS || sort==SORT_FPARS) {
         TContextViewNode   *child = node->GetChild(0);

         if(child->NodeHeight()) {
            child->Draw(dc, p);
            p += TSize(Indent, DistY+child->NodeHeight());
         }
      }
   }
```

```
    }
```

The variable `ActiveAddress` contains the TNodeAddress of the node for which the active context is to be shown. TNodeAddress is a class, based on an array that contains subsequent steps to take to go from the root node to the intended node. Using this class, the above functions inspect successive nodes along the root path (starting from the root of the tree). If a node is encountered whose first child is of sort DECS, PPARS or FPARS, this entire child and all its descendants are part of the active context.

6.4.3 TContextView class

The active context view class is created from the TScrolledTreeWindowView template class by supplying the view base class, TContextViewNode:

```
typedef TScrolledTreeWindowView<TContextViewNode> TContextView;
```

As explained in 6.2.3, the response table definition cannot be part of a class template and must be supplied separately for each view class. Unlike the Listing View and flow diagram view, the active context view needs not respond to menu commands and mouse clicks.

There are only two events (both of which are view notifications, sent by TStructDocument) that the context view must process:

- a change in the term tree made by the editor
- a request to show the active context for a different node

As a result, its response table is very small.

```
DEFINE_RESPONSE_TABLE1(TContextView, TScrolledWindowView)
   EV_VN_MIRRORSUBTREE,
   EV_VN_NEWACTIVE,
END_RESPONSE_TABLE;
```

The following function overrides the member function `StaticName` that would normally be generated by the TScrolledTreeWindowView template class. It returns a string: the name of the view. This string is used by the document manager to show a menu that lists the view types to choose from when adding a view.

```
const char far* TScrolledTreeWindowView<TContextViewNode>::StaticName()
{
   return "Context View";
}
```

The OWL macro below defines a template class TContextTemplate by associating the Document class TStructDocument with the view class TContextView.

```
DEFINE_DOC_TEMPLATE_CLASS(TStructDocument, TContextView, TContextTemplate);
```

The object `contextTpl` is an instance of TContextTemplate. This is the OWL way of registering a combination of Document class and View class with the Document Manager.

```
TContextTemplate contextTpl("Active Context View", "*.STR", 0, "STR",
                             dtAutoDelete | dtUpdateDir |dtAutoOpen);
```

Here, the text description is “Context View”. The other parameters are the same as the ones given to the TListingTemplate constructor.

7 Creating signatures and views

This chapter explains how to adapt the multiple-view structure editor to a new signature and how to add new view types to an existing structure editor.

Adapting the editor to a new signature involves creating a *signature module*. The signature module of the PL4 example editor, for instance, is called “`SIGN_PL4`” (See Appendix B). A general recipe for creating a signature module is given. In addition, an editor needs at least one view type, so creating an editor for a new signature implies creating one or more view types as well.

To add a new view type to the editor, a *view module* needs to be developed. A view module is self-contained: it can be added to an existing editor without any recompilation of other modules.

7.1 Creating a signature module

A signature module consists of a header file (suffix `.h`) and a C++ source file (suffix `.cpp`).

The header file contains the following:

- an enumeration of the operator constants,
- the definitions of all sorts and holes,
- the definitions of ‘real’ operators,
- a function template for the ‘operator object factory’, `CreateOperatorNode`.

For the definition of sorts, holes and operators, the macros introduced in Section 4.1.3 are used.

The full signature header file pattern is:

```
enum Operator {
   OP_NONE = 0,
   OP_S1HOLE, OP_S2HOLE, ... , OP_S|S|HOLE,
   OP_G|S|+1, OP_G|S|+2, ..., OP_G|G|
};

DEFINE_SORT_AND_HOLE(Si)
...
DEFINE_SORT_AND_HOLE(S|S|)
```

The following line must be repeated for all aggregate operators $\mathrm{g}:\{f_1:s_1,\ldots,f_n:s_n\}\rightarrow s_0$ *(not holes) where n is the operator arity:*

```
DEFINE_OPn(g, s0, f1, s1, ..., fn, sn)
```

The following line must be repeated for all list operators $\mathrm{g}:\{f_1:s_1{}^*\}\rightarrow s_0$*:*

```
DEFINE_OP_LIST(g, s0, f1, s1)
```

s is the sort of the term to be edited in the structure editor:

```
const Sort    TERMSORT = SORT_s;

const Operator   OP_FIRSTHOLE   = OP_G1;
const Operator   OP_LASTHOLE    = OP_G|S|;
const Operator   OP_FIRST       = OP_G|S|+1;
const Operator   OP_LAST        = OP_G|G|;

template<class T>
T              *CreateOperatorNode(T *,const Operator op)
{
   T              *node = 0;
   switch(op) {
// for all operators, including holes:
   case OP_G1:      node = new TG1<T>;     break;
   ...
   case OP_G|G|:     node = new TG|G|<T>;   break;
   }
   return node;
}
```

The file “TERM.h” contains the DEFINE_OP*n* macros for 0 **£** *n* **£** 5. If you have an aggregate operator with arity greater than 5, you need to add a corresponding macro to “TERM.h”. In all cases, you need to add the following line to “TERM.h” in order to attach your signature to the structure editor:

```
#include “signature module name.h”
```

A comment in “TERM.h” indicates where this line should be added.

The second part of the signature module is the signature source file (.cpp). It consists of the following functions:

`void    AskData(TWindow *window, TTerm *node)`

Allows signature implementer to create an operator that needs additional data. This function calls `node->GetOperator()`. If it is an operator that needs additional info, `AskData` should ask the user for the data. The window parameter makes it possible to construct a TInputDialog or other dialog boxes. This function is called by `TStructDocument::Refine`, see Section 5.2.

`Sort    GetOpSort(Operator op)`

Returns the operator result sort.

`char   *GetSortName(Sort sort)`

Returns the name of the given sort.

`char   *GetOpName(Operator op)`

Returns the name of operator `op`.

This is the signature source file pattern:

```
//---------------------------------------------------------------------------
//    Copyright
//      Implements ... signature
//---------------------------------------------------------------------------
#include "owlhdr.h"
#pragma hdrstop

#include "tree.h"
#include "strucdoc.h"


void      AskData(TWindow *window, TTerm *node)
{
   switch(node->GetOperator()) {
   case ...:
   case ...:
      node->SetData(data, data length);
      break;
   case ...:
   case ...:
      node->SetData(data, data length);
      break;
   }
}

Sort      GetOpSort(Operator op)
{
   if(op<OP_FIRST)
      return (Sort)op; // op is a hole <=> sort constant == op constant!
   switch(op) {

      // g ... h Î GS1
   case OP_g:
   ...
   case OP_h:              return SORT_S1;

   ...

      // j ... l Î GS|S|
   case OP_j:
   ...
   case OP_l:              return SORT_S|S|;

   }
   return SORT_NONE;
}


static char     *OpName[] = {
   “G|S|+1”, ..., “OP_G|G|”
};

static char     *SortName[] = {
   "",
   "S1", ... , "S|S|"
};

char   *GetSortName(Sort sort)
{
   return SortName[(int)sort];
}

char   *GetOpName(Operator op)
{
   if(!op)
      return 0;
   if(op<OP_FIRST) {
      static char     nbuf[MAXOPNAMELEN+1];
      wsprintf(nbuf, "%sHOLE", GetSortName((Sort)op));
      return nbuf;
   } else {
      return OpName[(int)(op-OP_FIRST)];
   }
}
```

This completes the entire signature definition.

7.2 Creating view modules

A view module can be created as a single source file (`.cpp`): no header file is required by other parts of the structure editor.

This section discusses creating a new view type, based on TScrolledTreeWindowView, the basic view user interface (Section 5.3.3). It is perfectly possible to write a view from scratch, but then there are so many degrees of freedom that it hardly makes sense to attempt to give a recipe for this case.

A view source file for a view based on TScrolledTreeWindowView has these components:

- Declaration of view base class **b** (derived from TViewNode) plus definition of member functions.
- For every operator **g** in the signature (including holes) a `CalcSize` and a `Draw` function.

  ```
  void  Tg<b>::CalcSize(TMyDC& dc)
  {    ... }
  void  Tg<b>::Draw(TMyDC& dc, TPoint& pos)
  {    ... }
  ```

  Please refer to Chapter 4 for an explanation of these functions.
- A typedef for the view type
- A response table
- Any functions in the TScrolledTreeWindowView that need a special version for this view type. At least: `TScrolledTreeWindowView<b>::StaticName()` which returns the name of the view. This name is used by the document manager to show a menu that lists the view types to choose from when adding a view.
- A Paint function (to override the null default TWindow paint function).

- Definition of a *document template class*, which serves to associate a document class (in the structure editor case: TStructDocument) with the view class defined in this module.
- At least one instantiation of the document template class, which lets OWL register this combination of Document class and View class with the Document Manager.

  The constructor parameters are: Text description of template, file name filter, default directory (0 indicates current directory), file extension, flags.

  Flags values often used:

`dtAutoDelete`:	Close and delete the document object when the last view is closed
`dtUpdateDir`:	Update directory with file dialog directory
`dtAutoOpen`:	Open a document upon creation

  Refer to the OWL documentation for a listing of all available `dtxxx` flags.

If the new view needs to control the operator menu strings, the view class should be a derived class of `TScrolledTreeWindowView<`**`b`**`>` that overrides the standard `GetOpMenuString` member function:

```
char *GetOpMenuString(Operator op)
{
   // return menu string for operator op
}
```

The same holds for accelerator keys: there is a member function `GetOpKey` which, by default, returns 0 (no key for this operator).

```
char     GetOpMenuString(Operator op)
{
   // return accelerator keyboard character for operator op
}
```

This is the view module source file pattern:

```
//---------------------------------------------------------------------------
//    Copyright
//      Implements class TMyView
//---------------------------------------------------------------------------
#include "owlhdr.h"
#pragma hdrstop

#include <owl\menu.h>

#include "views.rc"
#include "tree.h"
#include "strucdoc.h"
#include "mydc.h"
#include "viewnode.h"
#include "viewtree.h"
#include "scrwview.h"
#include "scrtrwvw.h"



class   b: public TViewNode {
public:
                      b(int n = 0, int fixed = FALSE): TViewNode(n, fixed) {}
   inline b         *GetChild(int n) const;
   ...
};

inline b* b::GetChild(int n) const
{  return (b*)TTreeNode::GetChild(n);
}



void  Tg<b>::CalcSize(TMyDC& dc)
{    ... }
void  Tg<b>::Draw(TMyDC& dc, TPoint& pos)
{    ... }


typedef TScrolledTreeWindowView<b> TMyView;
```

This is the full response table:

```
DEFINE_RESPONSE_TABLE1(TMyView, TScrolledWindowView)
   EV_WM_CHAR,
   EV_WM_LBUTTONDOWN,
   EV_WM_RBUTTONDOWN,
   EV_WM_LBUTTONDBLCLK,
   EV_WM_KEYDOWN,
   EV_WM_KEYUP,
   EV_WM_SETFOCUS,
   EV_VN_MIRRORSUBTREE,
   EV_VN_GLOBALFOCUS,

   EV_COMMAND(CM_PREVNODE,       PrevNode),
   EV_COMMAND(CM_NEXTNODE,       NextNode),
   EV_COMMAND(CM_PREVHOLE,       PrevHole),
   EV_COMMAND(CM_NEXTHOLE,       NextHole),
   EV_COMMAND(CM_DELETE,         Delete),
   EV_COMMAND(CM_CUT,            Cut),
   EV_COMMAND(CM_COPY,           Copy),
   EV_COMMAND(CM_PASTE,          Paste),
   EV_COMMAND(CM_PARENT,         Parent),
   EV_COMMAND(CM_FOLD,           ToggleFold),
   EV_COMMAND(CM_ADDHOLETOLIST,  AddHoleToList),
   EV_COMMAND_AND_ID(CM_ADDHOLEBEFORE,  AddHoleBeforeAfter),
   EV_COMMAND_AND_ID(CM_ADDHOLEAFTER,   AddHoleBeforeAfter),
END_RESPONSE_TABLE;

const char far* TScrolledTreeWindowView<b>::StaticName()
{
   return "My View";
}

void TListingView::Paint(TDC& dc, BOOL, TRect&)
{
   dc.SetBkMode(TRANSPARENT);
   if(Tree.GetRoot()) {
      Tree.GetRoot()->FlushNodePositions();
      Tree.GetRoot()->Draw((TMyDC&)dc, TPoint(0, 0));
   }
}


DEFINE_DOC_TEMPLATE_CLASS(TStructDocument, TMyView, TMyTemplate);
TMyTemplate myTpl("My View", "*.STR", 0, "STR",
                     dtAutoDelete | dtUpdateDir |dtAutoOpen | dt_xxxx ...);
```

Once a view module has been defined this way, it can be compiled and linked together with all other structure editor modules.

8 Conclusions

The most important general conclusion is: the algebraic framework about signatures and **S**-algebras has been an extremely great help in creating the Multiple-View Structure Editor design.

- For all operators **g** in the signature that have result sort *s* (the elements of the set $\mathbf{G}_s$), the C++ class for *s* is the direct base class of the C++ class for **g**. A pointer to a **g** object is at the same time a pointer to an *s* object. By using inheritance, the commonality between operators of the same result sort is expressed in C++ .
- In the mathematical discussion we have seen that a signature **S** captures the structural properties of all **S**-algebras derived from it. I have discovered how C++ template classes could be used to model a signature **S**. These C++ template classes are only defined once, yet they are used to define the structure of the C++ models of several **S**-algebras (**S**-term algebras and **S**-view algebras).
- The homomorphism concept has been translated quite literally to C++. In the structure editor the homomorphism from **S**-term algebra to **S**-view algebras is used to create a view tree from an existing term tree. For each view type, a separate homomorphism exists.
- The concept of a **S**-view algebra ($ViewTerm_{\mathbf{S},I,D}$, $F(I,D)$) captures most of what is involved in creating a graphical representation of a term tree. The set *I* is modelled by the *view base class* used. The functions *D* are modelled in C++ by a `CalcSize` function for every operator. Polymorphism is used in the implementation to make sure that the correct `CalcSize` function is called for a given object. The functions *D* calculate the *I* values associated with each node in the view tree in a bottom-up manner. It is possible to calculate a graphical representation of the view tree this way: A *D* function could compute a graphical representation ("pixel block") by combining the graphical representations from all child nodes and adding its own decorations. By doing this bottom-up, the root node eventually holds a graphical representation of the entire view tree. This is very unpractical however and very memory-inefficient because it involves a pixel block for each view tree node.

  To avoid this, the `Draw` functions were created, which have no counterpart in the mathematical framework. Their purpose is to create the graphical representation of a node and its descendants by drawing directly to the screen. Creating a graphical representation is a *side effect* of these functions: no "pixel blocks" are returned to the caller of a `Draw` function. Each operator has corresponding `Draw` functions, one for each view type, that "know" how the operator should be graphically represented.

The starting point for this work has been a structure editor, created by Mr.Hemerik, in Borland PASCAL with Objects. The application framework used here was TurboVision (for DOS). The structure editor developed in this thesis is implemented in C++, using ObjectWindows Library (for MS-Windows). The conclusions related to the implementation environment used are:

- C++ has templates classes and functions, PASCAL with Objects does not. The C++ structure editor implementation uses this language feature when defining the signature. Each sort and each operator in the signature is modelled by one template class. An operator template class is used for creating tree nodes of both term trees and view trees: it is not necessary to have a special version of the signature definition for each view type.

- The C++ template feature has made it possible to define all signature-related types, constants and functions in a single module, no matter how many view types there are. To create a structure editor for another signature, one only needs to replace the signature module by a new one. The procedure for creating signature modules is given in Chapter 7.
- The move from the character-oriented TurboVision user interface to ObjectWindows Library / MS-Windows has had both positive and negative aspects. TurboVision has TView and TGroup classes that allow a programmer to define (groups of) movable, resizable rectangles on the screen. The original structure editor uses these classes to offer the possibility to move and resize subterms on the screen. OWL does not have counterparts to TurboVision's TView and TGroup classes. It will therefore take more effort to implement similar functionality in the C++ structure editor.

The advantage of MS-Windows is that it is a graphical user interface. It offers much more freedom in designing graphical representations of terms. In the PL4 case, this has made it possible to create the flow diagram view. By using scroll bars, the Windows editor can handle graphical term representations that extend beyond the limits of the view window.

The Windows MDI user interface standard fits the requirement of having multiple views on a term, and having multiple editors (for working on multiple terms simultaneously) quite well. View windows can be conveniently resized, opened, closed and iconified.

- The OWL Doc / View model provides a good framework for associating a term with a number of views. It has helped in reaching the goal of creating a program structure where the editor object does not know anything about the number and types of associated views. Views are completely self-contained program modules. When a view type is added to the editor, absolutely no change to the other structure editor source files is necessary. The recipe in Chapter 7 explains how to create a view module. A new view module can be simply compiled and linked to the existing editor modules.

In the Multiple-View Structure Editor design, the design decisions were mainly based on:

- Algebraic theoretical framework of signatures and **S**-algebras.
- The PASCAL version used as a starting point.
- The object oriented implementation programming language C++.
- The application framework (OWL) and the target environment (MS-Windows).

The structure editor is still far from finished and it is very hard to predict how the design choices made for the current version will work out when new features are added.

Some suggestions for future research are:

- Addition of a formatting algorithm to the views in the structure editor. This means that there will be a number of different ways to graphically represent an operator. A simple example is the PL4 IFTHEN operator where the following two ways can be distinguished:

  The horizontal way

  ```
  if <?EXPR> then <?STATS> fi
  ```

  and the vertical way

  ```
  if <?EXPR> then
     <?STATS>
  fi
  ```

  Either the user or the system can decide which of the available representations to choose. A way to implement a decision by the system is to extend the `CalcSize` functions. Based on the relative sizes of child nodes, the available space on the screen and other information, it can choose the best representation.

In the case of the PL4 IFTHEN operator, the `CalcSize` function could do the following: if height of STATS child is 1 character or less and the total width of expression, decorations and statements does not exceed the screen width, choose the horizontal representation. Otherwise, choose the vertical representation.

- In the PL4 example editor, entering expressions is a tedious process. To improve the usability, it would be a good idea to offer the possibility to temporarily switch to a text editor mode. In this mode, a text editor will show the currently focused subtree in text form. This text form may then be edited, and on switching back, the text is parsed. Then, the subtree resulting from the parse process is connected to the term tree.
- Checking of context conditions
- Making a connection between terms of a different signature, e.g. lambda calculus and GCL, for constructing program and proof simultaneously.

Bibliography

[1] B. Meyer
Introduction to the Theory of Programming Languages
Prentice Hall, 1990

[2] C.M. de Deugd
Master's thesis: *A Mathematical Framework for Image Construction in Syntax-Directed Editors*
Eindhoven Technical University, 1992

[3] B.W. Watson
Computing Science Note 93/43. A taxonomy of finite automata construction algorithms
Eindhoven Technical University, December 1993

[4] dr. Ten Eikelder, dr. ir. C. Hemerik
Compilers 1 lecture notes
Eindhoven Technical University

[5] B. Stroustrup
The C++ Programming Language, Second Edition
Addison-Wesley Publishing Company, 1991

[6] Borland International, Inc.
Borland ObjectWindows for C++V2.0 Programmer's Guide
Borland ObjectWindows for C++V2.0 Library Reference
1993

Acknowledgements

I want to thank my supervisors, dr. ir. C. Hemerik and B.W. Watson, for their outstanding guidance and support. Their suggestions, ideas and constructive criticism have been a great help in realizing the Multiple-View Structure Editor and this master's thesis.

I am grateful to Sarah Lindemann for her love and support and for using her exceptional language skills in correcting the 'Eindhoven Scrolls'.

Hugo Lyppens
Eindhoven, April 1995

A Structure Editor Source Code

This appendix contains the entire Structure Editor source code (without signature and view modules). The following table lists all source and header files and their contents:

File	C++ Classes	Description
`main.cpp`	WinStrucApp	• contains entry point OwlMain() • defines and implements application object • defines response table
`term.h`	TTerm	• defines the DEFINE_OPxxx macros • includes signature module header file • defines TTerm class
`term.cpp`	TTerm	• implements TTerm
`tree.h`	TTreeNode TNodeAddress TTree<**b**>	• class definitions for TTreeNode and TNodeAddress • defines full TTree class template
`tree.cpp`	TTreeNode	• implements TTreeNode
`editor.h`	TTermEditor	• definition
`editor.cpp`	TTermEditor	• implementation
`strucdoc.h`	TStructDocument	• definition of class • vnXXX view notification message constants
`strucdoc.cpp`	TStructDocument	• implementation
`viewnode.h`	TViewNode	• definition
`viewnode.cpp`	TViewNode	• implementation
`viewtree.h`	TViewTree<**b**>	• defines full TViewTree class template
`mydc.h`	TMyDC	• definition
`mydc.cpp`	TMyDC	• implementation
`scrwview.h`	TScrolledWindowView	• definition
`scrwview.cpp`	TScrolledWindowView	• implementation+response table
`scrtrwvw.h`	TScrolledTreeWindowView<**b**>	• defines class template
`owlhdr.h`		• includes ObjectWindows headers

main.cpp

```
/*  Project WinStruc
    Copyright 1995 Hugo Lyppens
*/


#include "owlhdr.h"
#pragma hdrstop

#include "main.rc"            // Definition of all
resources.


class WinStrucApp : public TApplication {
private:

public:
   WinStrucApp ();
   virtual ~WinStrucApp ();

   TMDIClient  *mdiClient;

//{{WinStrucAppVIRTUAL_BEGIN}}
public:
   virtual void InitMainWindow();

protected:
   void EvNewView (TView& view);
   void EvCloseView (TView& view);
   DECLARE_RESPONSE_TABLE(WinStrucApp);
};




DEFINE_RESPONSE_TABLE1(WinStrucApp, TApplication)
    EV_OWLVIEW(dnCreate, EvNewView),
    EV_OWLVIEW(dnClose,  EvCloseView),
END_RESPONSE_TABLE;


WinStrucApp::WinStrucApp () : TApplication("Multi-View
Structure Editor")
{

    DocManager = new TDocManager(dmMDI | dmMenu);
}


WinStrucApp::~WinStrucApp ()
{
}


void WinStrucApp::InitMainWindow ()
{
    mdiClient = new TMDIClient;
    TDecoratedMDIFrame* frame = new
TDecoratedMDIFrame(Name, MDI_MENU, *mdiClient, TRUE);

    nCmdShow = (nCmdShow != SW_SHOWMINNOACTIVE) ?
SW_SHOWNORMAL : nCmdShow;

    //
    // Assign ICON w/ this application.
    //
    frame->SetIcon(this, IDI_MDIAPPLICATION);

    //
    // Menu associated with window and accelerator table
associated with table.
    //
    frame->AssignMenu(MDI_MENU);
    //
    // Associate with the accelerator table.
    //
//  frame->Attr.AccelTable = MDI_MENU;



    TStatusBar *sb = new TStatusBar(frame,
   TGadget::Recessed,
   TStatusBar::CapsLock        |
   TStatusBar::NumLock         |
   TStatusBar::ScrollLock      |
   TStatusBar::Overtype);
    frame->Insert(*sb, TDecoratedFrame::Bottom);

    SetMainWindow(frame);

}


////////////////////////////////////////////////////////
//
// WinStrucApp
// =====
// Response Table handlers:
//
void WinStrucApp::EvNewView (TView& view)
{
    TMDIClient *mdiClient =
TYPESAFE_DOWNCAST(GetMainWindow()->GetClientWindow(),
TMDIClient);
    if (mdiClient) {
        TMDIChild* child = new TMDIChild(*mdiClient, 0,
view.GetWindow());

        child->SetIcon(this, IDI_DOC);

        child->Create();
    }
}


void WinStrucApp::EvCloseView (TView&)
{
}


////////////////////////////////////////////////////////
//
// WinStrucApp
// ===========
// Menu Help About WinStruc.exe command
#if 0
void WinStrucApp::CmHelpAbout ()
{
    //
    // Show the modal dialog.
    //
    WinStrucAboutDlg(MainWindow).Execute();
}
#endif

int OwlMain (int , char* [])
{
    WinStrucApp     App;
    int             result;

    result = App.Run();

    return result;
}
```

term.h

```
#ifndef TERM_H
#define TERM_H

#include "tree.h"
#include "mydc.h"

typedef unsigned Sort;
const Sort SORT_NONE = 0;


#define DEFINE_OP0(name, rsort)  \
template<class T> \
class    T##name:  public TSort##rsort<T> { \
public: \
   Operator          GetOperator() { return OP_##name;
} \
   void                 Draw(TMyDC&, TPoint&); \
   void                 CalcSize(TMyDC&); \
\
                         T##name(); \
};\
template<class T> \
                         T##name<T>::T##name():
TSort##rsort<T>(0, TRUE) {}


#define DEFINE_OP1(name, rsort, a0name, a0sort)  \
template<class T> \
class    T##name:  public TSort##rsort<T> { \
public: \
   Operator          GetOperator() { return OP_##name;
} \
   void                 Draw(TMyDC&, TPoint&); \
   void                 CalcSize(TMyDC&); \
\
   TSort##a0sort<T>*    Get##a0name()  { return
(TSort##a0sort<T>*)GetChild(0); } \
   Sort                 GetDesiredChildSort(int n); \
   int                  GetDesiredNumChildren() { return
1; } \
\
                         T##name(); \
}; \
template<class T> \
                         T##name<T>::T##name():
TSort##rsort<T>(1, TRUE) {} \
\
template<class T> \
Sort              T##name<T>::GetDesiredChildSort(int n)
\
{ \
   switch(n) { case 0:  return SORT_##a0sort; \
               default: return SORT_NONE; \
   } \
}



#define DEFINE_OP2(name, rsort, a0name, a0sort, a1name,
a1sort)  \
template<class T> \
class    T##name:  public TSort##rsort<T> { \
public: \
   Operator          GetOperator() { return OP_##name;
} \
   void                 Draw(TMyDC&, TPoint&); \
   void                 CalcSize(TMyDC&); \
\
   TSort##a0sort<T>*    Get##a0name()  { return
(TSort##a0sort<T>*)GetChild(0); } \
   TSort##a1sort<T>*    Get##a1name()  { return
(TSort##a1sort<T>*)GetChild(1); } \
   Sort                 GetDesiredChildSort(int n); \
   int                  GetDesiredNumChildren() { return
2; } \
\
                         T##name(); \
}; \
template<class T> \
                         T##name<T>::T##name():
TSort##rsort<T>(2, TRUE) {} \
\
template<class T> \
Sort              T##name<T>::GetDesiredChildSort(int n)
\
{ \
   switch(n) { case 0:  return SORT_##a0sort; \
               case 1:  return SORT_##a1sort; \
               default: return SORT_NONE; \
   } \
}



#define DEFINE_OP3(name, rsort, a0name, a0sort, a1name,
a1sort, a2name, a2sort)  \
template<class T> \
class    T##name:  public TSort##rsort<T> { \
public: \
   Operator          GetOperator() { return OP_##name;
} \
   void                 Draw(TMyDC&, TPoint&); \
   void                 CalcSize(TMyDC&); \
\
   TSort##a0sort<T>*    Get##a0name()  { return
(TSort##a0sort<T>*)GetChild(0); } \
   TSort##a1sort<T>*    Get##a1name()  { return
(TSort##a1sort<T>*)GetChild(1); } \
   TSort##a2sort<T>*    Get##a2name()  { return
(TSort##a2sort<T>*)GetChild(2); } \
   Sort                 GetDesiredChildSort(int n); \
   int                  GetDesiredNumChildren() { return
3; } \
\
                         T##name(); \
}; \
template<class T> \
                         T##name<T>::T##name():
TSort##rsort<T>(3, TRUE) {} \
\
template<class T> \
Sort              T##name<T>::GetDesiredChildSort(int n)
\
{ \
   switch(n) { case 0:  return SORT_##a0sort; \
               case 1:  return SORT_##a1sort; \
               case 2:  return SORT_##a2sort; \
               default: return SORT_NONE; \
   } \
}



#define DEFINE_OP4(name, rsort, a0name, a0sort, a1name,
a1sort, a2name, a2sort, a3name, a3sort)  \
template<class T> \
class    T##name:  public TSort##rsort<T> { \
public: \
   Operator          GetOperator() { return OP_##name;
} \
   void                 Draw(TMyDC&, TPoint&); \
   void                 CalcSize(TMyDC&); \
\
   TSort##a0sort<T>*    Get##a0name()  { return
(TSort##a0sort<T>*)GetChild(0); } \
   TSort##a1sort<T>*    Get##a1name()  { return
(TSort##a1sort<T>*)GetChild(1); } \
   TSort##a2sort<T>*    Get##a2name()  { return
(TSort##a2sort<T>*)GetChild(2); } \
   TSort##a3sort<T>*    Get##a3name()  { return
(TSort##a3sort<T>*)GetChild(3); } \
   Sort                 GetDesiredChildSort(int n); \
   int                  GetDesiredNumChildren() { return
4; } \
\
                         T##name(); \
}; \
template<class T> \
                         T##name<T>::T##name():
TSort##rsort<T>(4, TRUE) {} \
\
\
template<class T> \
Sort              T##name<T>::GetDesiredChildSort(int n)
\
{ \
   switch(n) { case 0:  return SORT_##a0sort; \
               case 1:  return SORT_##a1sort; \
               case 2:  return SORT_##a2sort; \
               case 3:  return SORT_##a3sort; \
               default: return SORT_NONE; \
   } \
}



#define DEFINE_OP5(name, rsort, a0name, a0sort, a1name,
a1sort, a2name, a2sort, a3name, a3sort, a4name, a4sort)
\
template<class T> \
class    T##name:  public TSort##rsort<T> { \
public: \
   Operator          GetOperator() { return OP_##name;
} \
   void                Draw(TMyDC&, TPoint&); \
   void                CalcSize(TMyDC&); \
\
   TSort##a0sort<T>*    Get##a0name()  { return
(TSort##a0sort<T>*)GetChild(0); } \
   TSort##a1sort<T>*    Get##a1name()  { return
(TSort##a1sort<T>*)GetChild(1); } \
   TSort##a2sort<T>*    Get##a2name()  { return
(TSort##a2sort<T>*)GetChild(2); } \
   TSort##a3sort<T>*    Get##a3name()  { return
(TSort##a3sort<T>*)GetChild(3); } \
   TSort##a4sort<T>*    Get##a4name()  { return
(TSort##a4sort<T>*)GetChild(4); } \
   Sort                GetDesiredChildSort(int n); \
   int                 GetDesiredNumChildren() { return
5; } \
\
                        T##name(); \
}; \
template<class T> \
                        T##name<T>::T##name():
TSort##rsort<T>(5, TRUE) {} \
\
\
template<class T> \
Sort             T##name<T>::GetDesiredChildSort(int n)
\
{ \
   switch(n) { case 0:  return SORT_##a0sort; \
               case 1:  return SORT_##a1sort; \
               case 2:  return SORT_##a2sort; \
               case 3:  return SORT_##a3sort; \
               case 4:  return SORT_##a4sort; \
               default: return SORT_NONE; \
   } \
}




#define DEFINE_OP_LIST(name, rsort, aname, asort)  \
template<class T> \
class    T##name:  public TSort##rsort<T> { \
public: \
   Operator          GetOperator() { return OP_##name;
} \
   void                Draw(TMyDC&, TPoint&); \
   void                CalcSize(TMyDC&); \
\
   TSort##asort<T>*     Get##aname(int n)  { return
(TSort##asort<T>*)GetChild(n); } \
   Sort                GetDesiredChildSort(int n) {
return n>=0?SORT_##asort:SORT_NONE; } \
   BOOL                IsList() { return TRUE; } \
\
                        T##name(); \
};\
template<class T> \
                        T##name<T>::T##name():
TSort##rsort<T>(0, FALSE) {} \
\


#define DEFINE_SORT(name)  \
const Sort SORT_##name = OP_##name##HOLE;\
template<class T> \
class    TSort##name: public T { \
public: \
   Sort              GetSort() { return SORT_##name; }
\
                        TSort##name(int n = 0, int
fixed = FALSE); \
};\
template<class T> \
                        TSort##name<T>::TSort##name(int
n, int fixed): T(n, fixed) {}

#define DEFINE_SORT_AND_HOLE(name)  \
DEFINE_SORT(name) \
DEFINE_OP0(name##HOLE,name)


// insert '#include "my signature module .h"' here!

#include "sign_pl4.h"
```

```
const        MAXOPNAMELEN   = 20;
const        MAXOPDATALEN   = 100;

char        *GetSortName(Sort sort);
char        *GetOpName(Operator op);
Sort         GetOpSort(Operator op);



class    TTerm: public TTreeNode {
public:
   inline         TTerm(int n = 0, int fixed = FALSE):
TTreeNode(n, fixed),Data(0),Length(0) {}
                    ~TTerm();

   virtual Operator  GetOperator() = 0;
   inline char      *GetOpName()               {
return ::GetOpName(GetOperator()); }
   virtual int       GetDesiredNumChildren()    { return
0; }
   virtual BOOL      IsList()                  { return
FALSE; }

   virtual Sort      GetSort()      = 0;
   inline char      *GetSortName()             {
return ::GetSortName(GetSort()); }

   TTerm            *GetChild(int n)           {
return (TTerm*)TTreeNode::GetChild(n);}
   virtual Sort      GetDesiredChildSort(int n) { return
SORT_NONE; }

   void              SetData(void *data, int len);
   void             *GetData()              { return
Data; }
   int               GetDataLength()         { return
Length; }

   int              GetTermLength();
   friend ostream&   operator<<(ostream&, TTerm *);
private:
   void             *Data;
   int               Length;
};

void     AskData(TWindow *window, TTerm *node);

#endif
```

term.cpp

```
//-------------------------------------------------------
---------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TTerm
//-------------------------------------------------------
---------------------
#include "owlhdr.h"
#pragma hdrstop
#include "term.h"



int              TTerm::GetTermLength()
{
    int     nc = GetNumChildren();
    int     tl = strlen(GetOpName())+(nc?nc+1:0);
    if(GetData())
      tl += 2+strlen((char *)GetData());

    for(int i = 0; i<nc; i++) {
        tl+=GetChild(i)->GetTermLength();
    }
    return tl;
}

ostream&         operator<<(ostream& s, TTerm *term)
{
    int     nc = term->GetNumChildren();

    s << GetOpName(term->GetOperator());
    if(term->GetData())
      s << '{' << (char *)(term->GetData()) << '}';
    if(nc) {
        s << '[';
        for(int i = 0; i<nc; i++) {
            s << term->GetChild(i);
            if(i!=nc-1)
                s << ',';
        }
        s << ']';
    }
    return s;
}

TTerm::~TTerm()
{
   if(Data)
      free(Data);
}

void TTerm::SetData(void *d, int length)
{
   if(Data)
      free(Data);
   Data = 0; Length = 0;
   if(d && length) {
      Data = malloc(length); Length = length;
      memcpy(Data, d, length);
   }
}
```

tree.h

```
#ifndef TREE_H
#define TREE_H


class TTreeNode {
public:
   inline                  TTreeNode(int n = 0, int
fixed = FALSE);
   virtual                   ~TTreeNode() {}
   inline TTreeNode          *GetChild(int n) const;
   inline TTreeNode          *GetParent() const;
   inline int                 GetNumChildren() const;
   void                       SetChildNumbers(int n);
   long                       CountSubtreeNodes() const;
   BOOL                       IsDescendant(TTreeNode
*anc) const;

   TTreeNode                 *Parent;
   int                        MyChildNumber;
   TArrayAsVector<TTreeNode*>  Children;
};

inline                     TTreeNode::TTreeNode(int n,
int fixed):

Children(n,0,fixed?0:2),Parent(0),MyChildNumber(0)
{
}

inline TTreeNode  *TTreeNode::GetChild(int n) const
{  return n>=0 && n<GetNumChildren()?Children[n]:0;
}

inline TTreeNode             *TTreeNode::GetParent()
const
{  return Parent;
}

inline int
TTreeNode::GetNumChildren() const
{  return Children.GetItemsInContainer();
}



class TNodeAddress : public TArrayAsVector<short> {
public:
   inline         TNodeAddress() :
TArrayAsVector<short>(9,0,10) {}

   inline int     Length() const { return
GetItemsInContainer(); }
   inline short GetChildNumber(int n) const {return
ItemAt(GetItemsInContainer()-1-n); }
};



template<class T>
class TTree {
public:
   inline           TTree():Root(0),NumNodes(0) {}
                      ~TTree();

   inline   T          *GetRoot() const;
   inline   long        GetNumNodes() const;
   inlineBOOL        IsLeaf(T *e) const;
   inlineBOOL        IsRoot(T *e) const;

   inlineint         GetMyChildNumber(T *e) const;
   inlineT          *GetParent(T *e)   const;
   inlineT          *GetChild(T *e, int n) const;
   inlineint         GetNumChildren(T *e) const;
   inlineT          *GetSibling(T *e, int n) const;
   inlineT          *GetLeftSibling(T *e) const;
   inlineT          *GetRightSibling(T *e) const;

   T                  *GetLastNode()     const;
   T                  *GetNextNode(T *e) const;
   T                  *GetPrevNode(T *e) const;

   inline   void        InsertChild(T *par,  int
childnumber, T *child);
   void                AddChild(T *par, T *child);
   void                ReplaceChild(T *par, int
childnumber, T *newchild, BOOL del=TRUE);
   void                RemoveChild(T *par, int
childnumber, BOOL del=TRUE);
```

```
   inline   void        RemoveSubtree(T *subtree, BOOL
del=TRUE);
   inline   void        ReplaceSubtree(T *subtree, T
*replacement, BOOL del=TRUE);

   void                 GetNodeAddress(T *node,
TNodeAddress& address) const;
   T                   *NodeAt(const TNodeAddress&
address) const;
private:
   void                 DoRecursive(T *e, void pre(T *),
void post(T *));

   T                   *Root;
   long                 NumNodes;
};

template <class T>
TTree<T>::~TTree()
{
   if(Root)
      RemoveChild(0, 0);
}
template <class T> inline T        *TTree<T>::GetRoot()
const
{  return Root;
}
template <class T> inline long
TTree<T>::GetNumNodes() const
{  return NumNodes;
}
template <class T> inline   BOOL     TTree<T>::IsLeaf(T
*e) const
{  return e->Children.IsEmpty();
}
template <class T> inline   BOOL     TTree<T>::IsRoot(T
*e) const
{  return e->Parent==Root;
}
template <class T> inline   int
TTree<T>::GetMyChildNumber(T *e) const
{  return e->MyChildNumber;
}
template <class T> inline   int
TTree<T>::GetNumChildren(T *e) const
{  return e?e-
>Children.GetItemsInContainer():(Root?1:0);
}
template <class T> inline   T
*TTree<T>::GetChild(T *e, int n) const
{  return (T*)(e?
     (n>=0 && n<e->Children.GetItemsInContainer() ? e-
>Children[n] : 0) :
     (n==0?Root:0));
}
template <class T> inline   T
*TTree<T>::GetSibling(T *e, int n) const
{  return GetChild(GetParent(e), n);
}
template <class T> inline   T
*TTree<T>::GetLeftSibling(T *e) const
{  return GetSibling(e, e->MyChildNumber-1);
}
template <class T> inline   T
*TTree<T>::GetRightSibling(T *e) const
{  return GetSibling(e, e->MyChildNumber+1);
}
template <class T> inline   T       *TTree<T>::
GetParent(T *e) const
{  return (T*)e->Parent;
}
template <class T> inline  void     TTree<T>::AddChild(T
*par, T *child)
{  InsertChild(par, GetNumChildren(par), child);
}
template <class T> inline  void
TTree<T>::RemoveSubtree(T *subtree, BOOL del)
{
RemoveChild(GetParent(subtree),GetMyChildNumber(subtree)
, del);
}
template <class T> inline  void
TTree<T>::ReplaceSubtree(T *subtree, T *replacement,
BOOL del)
{
ReplaceChild(GetParent(subtree),GetMyChildNumber(subtree
), replacement, del);
}
template <class T>
void             TTree<T>::ReplaceChild(T *par, int
childnumber, T *child, BOOL del)
{
   if(childnumber>=0 && childnumber<GetNumChildren(par))
{
      T *oldc      = GetChild(par, childnumber);
      if(oldc) {
         NumNodes -= oldc->CountSubtreeNodes();
         oldc->Parent = 0;
         if(del) {
            DoRecursive(oldc, 0, DestroyNode);
         }
      }
      if(par) {
         par->Children[childnumber] = child;
      } else if(!childnumber) {
         Root = child;
      }
      if(child) {
         child->Parent   = par; child->MyChildNumber =
childnumber;
         NumNodes    += child->CountSubtreeNodes();
      }
   }
}

template <class T>
void             TTree<T>::InsertChild(T *par, int
childnumber, T *child)
{
   if(par) {
      par->Children.AddAt(child, childnumber);
      par->SetChildNumbers(childnumber);
   } else if(!childnumber && !Root) {
      Root = child;
   } else
      return;
   if(child) {
      NumNodes             += child-
>CountSubtreeNodes();
      child->MyChildNumber  = childnumber;
      child->Parent         = par;
   }
}


template <class T>
void             TTree<T>::RemoveChild(T *e, int n,
BOOL del)
{
   ReplaceChild(e, n, 0, del);
   if(e) {
      e->Children.RemoveEntry(n); e->SetChildNumbers(n);
   } else if(!n) {
      Root = 0;
   }
}


template <class T>
void             DestroyNode(T *e)
{
   delete e;
}


template <class T>
void             TTree<T>::DoRecursive(T *e, void
pre(T*), void post(T*))
{
   unsigned   i, n = GetNumChildren(e);
   T         *c;

   if(pre) pre(e);
   for(i = 0; i<n; i++) {
      if(c = GetChild(e, i))
         DoRecursive(c, pre, post);
   }
   if(post) post(e);
}

template <class T>
void             TTree<T>::GetNodeAddress(T *node,
TNodeAddress& address) const
{
   address.Flush();
   while(node->Parent) {
      address.Add((short)(node->MyChildNumber));
      node = GetParent(node);
   }
}
```

```
template <class T>
T               *TTree<T>::NodeAt(const TNodeAddress&
address) const
{
   T    *node  = Root;

   for(int i = (int)address.GetItemsInContainer()-1;
i>=0; i--) {
      node = GetChild(node, address[i]);
   }
   return node;
}

template <class T>
T               *TTree<T>::GetLastNode() const
{
   T     *lastnode = Root;
   int    cn;

   while(cn = GetNumChildren(lastnode))
      lastnode = GetChild(lastnode, cn-1);
   return lastnode;
}

template <class T>
T               *TTree<T>::GetNextNode(T *e) const
{
   int    cn = 0;
   do {
      if(cn < GetNumChildren(e)) {
         return GetChild(e, cn);
      } else {
         cn = GetMyChildNumber(e)+1;
         e  = GetParent(e);
      }
   } while(e);
   return 0;
}



template <class T>
T               *TTree<T>::GetPrevNode(T *e) const
{
   T  *prevnode;

   prevnode = GetLeftSibling(e);
   if(!prevnode) {
      prevnode = GetParent(e);
   } else {
      int nc;
      while(nc = GetNumChildren(prevnode)) {
         prevnode = GetChild(prevnode, nc-1);
      }
   }
   return prevnode;
}


#endif
```

tree.cpp

```
//--------------------------------------------------------
----------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TTree/TTreeNode
//--------------------------------------------------------
----------------------
#include "owlhdr.h"
#pragma hdrstop
#include "tree.h"


void TTreeNode::SetChildNumbers(int n)
{
   for(int i = n; i<Children.GetItemsInContainer(); i++)
{
      Children[i]->MyChildNumber = i;
   }
}


long     TTreeNode::CountSubtreeNodes() const
{
   int   nc = GetNumChildren();
   int   nel = 1;

   for(int i = 0; i<nc; i++) {
      TTreeNode*  e = GetChild(i);
      if(e)
         nel += e->CountSubtreeNodes();
   }
   return nel;
}

BOOL     TTreeNode::IsDescendant(TTreeNode *anc) const
{
   const TTreeNode const  *p = this;
   while(p = p->GetParent()) {
      if(p == anc)
         return TRUE;
   }
   return FALSE;
}

void              DeleteTTreeNode(TTreeNode *e)
{
   delete e;
}
```

editor.h

```
#ifndef EDITOR_H
#define EDITOR_H

#include "term.h"

class TTermEditor: public TTree<TTerm>
{
public:
                    TTermEditor();
   BOOL             ReadSubtreeFromStream(istream& is,
TNodeAddress& addr);
   BOOL             ReadFromStream(istream& is);
   inline void      WriteSubtreeToStream(ostream& os,
TNodeAddress& addr);
   inline void      WriteToStream(ostream& os);

   void             Refine(TNodeAddress& address,
Operator op);
   void             Delete(TNodeAddress& address);
   void             ListInsert(TNodeAddress& address,
int childnumber);
   void             ListDelete(TNodeAddress& address,
int childnumber);

private:
   BOOL             ReadSubtreeFromStream(istream& is,
TTerm *parent, int cn);
   void             InsertHole(TTerm *parent, int cn);
   void             InsertOperator(TTerm *parent, int
cn, Operator op);
};


inline void TTermEditor::WriteSubtreeToStream(ostream&
os, TNodeAddress& addr)
{
   os << NodeAt(addr);
}

inline void TTermEditor::WriteToStream(ostream& os)
{
   if(GetRoot())
      os << GetRoot();
}
#endif
```

editor.cpp

```
//--------------------------------------------------------
---------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TTermEditor
//--------------------------------------------------------
---------------------
#include "owlhdr.h"
#pragma hdrstop
#include "mydc.h"
#include "tree.h"
#include "term.h"
#include "editor.h"



TTermEditor::TTermEditor()
{
   InsertHole(0, 0);
}

void     TTermEditor::InsertHole(TTerm *parent, int cn)
{
   Sort sort = parent ? parent->GetDesiredChildSort(cn)
: TERMSORT;
#ifdef AUTOOP
   int   number = 0;
   Operator opfound;
   for(Operator op = OP_FIRST; number <=1 &&
op<=OP_LAST; op++) {
      if(GetOpSort(op) == sort) {
         opfound = op; number++;
      }
   }
   if(number == 1) {
      InsertOperator(parent, cn, opfound);
      return;
   }
#endif
   TTerm    *hole = CreateOperatorNode((TTerm *)0,
      (Operator)sort);
   InsertChild(parent, cn, hole);
}

void     TTermEditor::InsertOperator(TTerm *parent, int
cn, Operator op)
{
   TTerm    *newnode = CreateOperatorNode((TTerm *)0,
op);
   InsertChild(parent, cn, newnode);
   int numchildholes = newnode->IsList() ? 1 : newnode-
>GetDesiredNumChildren();
   for(int i = 0; i<numchildholes; i++) {
      InsertHole(newnode, i);
   }
}


void     TTermEditor::Refine(TNodeAddress& address,
Operator op)
{
   TTerm    *oldnode, *parent;
   int       cn;

   oldnode = NodeAt(address);
   parent = GetParent(oldnode); cn =
GetMyChildNumber(oldnode);
   RemoveChild(parent, cn);
   InsertOperator(parent, cn, op);
}

void     TTermEditor::Delete(TNodeAddress& address)
{
   TTerm    *oldnode, *newnode;

   oldnode = NodeAt(address);
   newnode = CreateOperatorNode((TTerm *)0,
             (Operator)oldnode->GetSort());
   ReplaceSubtree(oldnode, newnode);
}



void     TTermEditor::ListInsert(TNodeAddress& address,
int childnumber)
{
   TTerm    *listnode, *childnode;

   listnode  = NodeAt(address);
   childnode = CreateOperatorNode((TTerm *)0,
         (Operator)listnode-
>GetDesiredChildSort(childnumber));
   InsertChild(listnode, childnumber, childnode);
}


void     TTermEditor::ListDelete(TNodeAddress& address,
int childnumber)
{
   RemoveChild(NodeAt(address), childnumber);
}

BOOL        TTermEditor::ReadFromStream(istream& is)
{
   static TNodeAddress rootaddr;

   return ReadSubtreeFromStream(is, rootaddr);
}

BOOL        TTermEditor::ReadSubtreeFromStream(istream&
is, TNodeAddress& addr)
{
   TTerm    *node = NodeAt(addr);
   TTerm *parent = GetParent(node);
   int       cn = GetMyChildNumber(node);

   RemoveChild(parent, cn);
   BOOL rv = ReadSubtreeFromStream(is, parent, cn);
   if(!rv) {
      InsertHole(parent, cn);
   }
   return rv;
}


BOOL        TTermEditor::ReadSubtreeFromStream(istream&
is, TTerm *parent, int cn)
{
   Operator       op;
   char           opname[MAXOPNAMELEN+1], *p = opname;
   static char    opdata[MAXOPDATALEN+1];
   char           c = 0;
   BOOL           found = FALSE;
   TTerm         *node;
   BOOL           rv = FALSE;
   BOOL           childinserted = FALSE;

   while(is && (c = (char)is.peek())!='[' && c!=']' &&
c!=',' && c!='{') {
      is >> *p; p++;
   }
   *p = 0;
   for(op = OP_FIRSTHOLE; op<= OP_LAST; op++) {
      found = !stricmp(opname, GetOpName(op));
      if(found)
         break;
   }
   if(!found)
      goto quit;
   node = CreateOperatorNode((TTerm *)0, op);
   if(!node)
      goto quit;
   if(c == '{') {
      is >> c;
      p = opdata;
      while(is >> c && c!='}') {
         *p++ = c;
      }
      *p++ = 0;
      node->SetData(opdata, p-opdata);
      c = (char)is.peek();
   }
   InsertChild(parent, cn, node);
   childinserted = TRUE;
   /* c ==  is.peek() */
   if(c == '[') {
      is >> c;
      do {
         if(!ReadSubtreeFromStream(is, node, node-
>GetNumChildren()))
            goto quit;
         c = 0;
      } while(is>>c && c==',');
      if(c!=']')
         goto quit;
   }
   if(!node->IsList() &&
       node->GetNumChildren() != node-
>GetDesiredNumChildren() )
            goto quit;
   rv = TRUE;
quit:
   if(!rv && childinserted)
      RemoveChild(parent, cn);
   return rv;
}
```

strucdoc.h

```
#ifndef STRUCDOC_H
#define STRUCDOC_H

#include "editor.h"

class  _DOCVIEWCLASS TStructDocument : public
TFileDocument
{
public:
    TStructDocument(TDocument* parent = 0);

    // implement virtual methods of TDocument
    BOOL           Open(int mode, const char far*
path=0);
//  BOOL           IsOpen() { return FALSE; }
    BOOL           Commit(BOOL force = FALSE);
    BOOL           Revert(BOOL clear = FALSE);

    // data access functions
   void            Refine(TNodeAddress& address,
Operator op, TWindow *window = 0);
   void            Delete(TNodeAddress& address);
   void            Cut(TNodeAddress& address, HWND);
   void            Copy(TNodeAddress& address, HWND);
   void            Paste(TNodeAddress& address, HWND);
   void            ListInsert(TNodeAddress& address,
int childnumber);
   void            ListDelete(TNodeAddress& address,
int childnumber);
   void            GlobalFocus(TNodeAddress& address);

   BOOL            ClipboardOK(const Sort sort, HWND
hwnd) const;
   TTree<TTerm>*GetTree() {return &Editor;}

   void            GetActiveAddress(TNodeAddress&
address) const;
   void            SetActiveAddress(TNodeAddress&
address);
private:
   TTerm          *ActiveNode;
   TTermEditor     Editor;
   static UINT     ClipboardFormat;
};


//////////////////////////////////////
const int vnMirrorSubtree = vnCustomBase+0;
const int vnGlobalFocus   = vnCustomBase+1;
const int vnNewActive  = vnCustomBase+2;

#endif
```

strucdoc.cpp

```
//----------------------------------------------------
---------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TStructDocument
//----------------------------------------------------
---------------------
#include "owlhdr.h"
#pragma hdrstop
#include "strucdoc.h"



UINT     TStructDocument::ClipboardFormat = 0;

TStructDocument::TStructDocument(TDocument* parent) :
TFileDocument(parent)
{
   SetDirty(FALSE);
   ActiveNode = Editor.GetRoot();
}


void     TStructDocument::Refine(TNodeAddress& address,
Operator op, TWindow *window)
{
   Editor.Refine(address, op);
   AskData(window, Editor.NodeAt(address));
   ActiveNode = Editor.NodeAt(address);
   SetDirty(TRUE);
   NotifyViews(vnMirrorSubtree, (long)&address);
}


void     TStructDocument::Delete(TNodeAddress& address)
{
   Editor.Delete(address);
   ActiveNode = Editor.NodeAt(address);

   SetDirty(TRUE);
   NotifyViews(vnMirrorSubtree, (long)&address);
}

void     TStructDocument::Cut(TNodeAddress& address,
HWND hwnd)
{
   Copy(address, hwnd);
   Delete(address);
}
void     TStructDocument::Copy(TNodeAddress& address,
HWND hwnd)
{
   TTerm      *node    = Editor.NodeAt(address);
   int         memlen  = node->GetTermLength()+1;
   HGLOBAL     handle  = 0, handle2 = 0;
   char       *p  = 0;
   Sort    *p2 = 0;
   TClipboard  cb(hwnd);

   handle  = ::GlobalAlloc(GHND, memlen);
   if(!handle)
      goto quit;
   handle2 = ::GlobalAlloc(GHND, sizeof(Sort));
   if(!handle2)
      goto quit;
   p = (char *)::GlobalLock(handle);
   if(!p)
      goto quit;
   p2 = (Sort *)::GlobalLock(handle2);
   if(!p2)
      goto quit;

   if(!ClipboardFormat) {
      ClipboardFormat =
cb.RegisterClipboardFormat("WinStruc Term Sort");
   }
   if(cb.EmptyClipboard()) {
      ostrstream  os(p, memlen);
      Editor.WriteSubtreeToStream(os, address);
      os << '\0';  // write terminating 0
      *p2 = node->GetSort();

      ::GlobalUnlock(handle);
      ::GlobalUnlock(handle2);
      cb.SetClipboardData(CF_TEXT,         handle);
      cb.SetClipboardData(ClipboardFormat, handle2);
      handle = 0; handle2 = 0; p = 0; p2 = 0;
   }
quit:
   if(p)
      ::GlobalUnlock(handle);
   if(p2)
      ::GlobalUnlock(handle2);
   if(handle)
      GlobalFree(handle);
   if(handle2)
      GlobalFree(handle2);
}
void     TStructDocument::Paste(TNodeAddress& address,
HWND hwnd)
{
   TTerm       *node    = Editor.NodeAt(address);
   HGLOBAL      handle, handle2;
   char        *p = 0;
   Sort     *p2 = 0;
   TClipboard   cb(hwnd);

   if(!ClipboardFormat)
      goto quit;
   handle2 = cb.GetClipboardData(ClipboardFormat);
   if(!handle2)
      goto quit;
   handle  = cb.GetClipboardData(CF_TEXT);
   if(!handle)
      goto quit;
   p2 = (Sort *)::GlobalLock(handle2);
   if(!p2 || *p2 != node->GetSort())
      goto quit;
   p  = (char *)::GlobalLock(handle);
   if(!p)
      goto quit;
   {
      istrstream  is(p);
      Editor.ReadSubtreeFromStream(is, address);
   }
   SetDirty(TRUE);
   ActiveNode = Editor.NodeAt(address);
   NotifyViews(vnMirrorSubtree, (long)&address);
quit:
   if(p)
      ::GlobalUnlock(handle);
   if(p2)
      ::GlobalUnlock(handle2);
}

BOOL     TStructDocument::ClipboardOK(const Sort sort,
HWND hwnd) const
{
   BOOL     IsOK = FALSE;
   if(ClipboardFormat) {
      TClipboard  cb(hwnd);
      HGLOBAL     handle2;

      handle2 = cb.GetClipboardData(ClipboardFormat);
      if(handle2) {
         Sort *p2 = (Sort *)::GlobalLock(handle2);
         if(p2) {
            if(*p2 == sort)
               IsOK = TRUE;
            ::GlobalUnlock(handle2);
         }
      }
   }
   return IsOK;
}


void     TStructDocument::ListInsert(TNodeAddress&
address, int childnumber)
{
   Editor.ListInsert(address, childnumber);
   SetDirty(TRUE);
   NotifyViews(vnMirrorSubtree, (long)&address);
}


void     TStructDocument::ListDelete(TNodeAddress&
address, int childnumber)
{
   Editor.ListDelete(address, childnumber);
   SetDirty(TRUE);
   ActiveNode = Editor.NodeAt(address);
   NotifyViews(vnMirrorSubtree, (long)&address);
}

void     TStructDocument::GlobalFocus(TNodeAddress&
address)
{
   SetActiveAddress(address);
   NotifyViews(vnGlobalFocus, (long)&address);
}



BOOL TStructDocument::Commit(BOOL force)
{
   if (!IsDirty() && !force)
      return TRUE;

   TOutStream* os = OutStream(ofWrite);
   if (!os)
      return FALSE;

   Editor.WriteToStream(*os);

   delete os;

   SetDirty(FALSE);
   return TRUE;
}

BOOL TStructDocument::Revert(BOOL clear)
{
   if (!TFileDocument::Revert(clear))
      return FALSE;
   if (!clear)
      Open(0, 0);
   static TNodeAddress rootaddr;
   ActiveNode = Editor.GetRoot();
   NotifyViews(vnMirrorSubtree, (long)&rootaddr);
   return TRUE;
}

BOOL   TStructDocument::Open(int /*mode*/, LPCSTR path)
{
   if (path)
      SetDocPath(path);
   if (GetDocPath()) {
      TInStream* is = InStream(ofRead);
      if (!is)
         return FALSE;

      Editor.ReadFromStream(*is);

      delete is;
   }
   ActiveNode = Editor.GetRoot();
   SetDirty(FALSE);
   return TRUE;
}

void
TStructDocument::GetActiveAddress(TNodeAddress& address)
const
{
   Editor.GetNodeAddress(ActiveNode, address);
}
void
TStructDocument::SetActiveAddress(TNodeAddress& address)
{
   TTerm    *newactive = Editor.NodeAt(address);

   if(ActiveNode != newactive) {
      ActiveNode = newactive;
      NotifyViews(vnNewActive, (long)&address);
   }
}
```

viewnode.h

```
class   TViewNode : public TTerm {
public:
                        TViewNode(int n = 0, int fixed
= FALSE);
   inline TViewNode    *GetChild(int n) const;
   inline TViewNode    *GetParent() const;

   inline BOOL          GetFolded() const;
   inline void          SetFolded(BOOL folded);
   inline TSize&        NodeSize();
   inline int           NodeHeight() const;
   inline int           NodeWidth() const;


   virtual void         Draw(TMyDC&, TPoint&) = 0;
   virtual void         CalcSize(TMyDC&)      = 0;

   void                 SetNodeSize(TSize& sz);
   void                 AddNodePosition(TPoint& pt);
   TPoint&              GetNodePosition(int n) const;
   int                  GetNumNodePositions() const;

   void                 FlushChildrenNodePositions();
   void                 FlushNodePositions();
   TViewNode           *WhichNode(TPoint& point);
   BOOL                 IsVisible() const;
private:
   TSize                Size;
   BOOL                 Folded;
   TArrayAsVector<TPoint>   NodePositions;
};


inline TViewNode* TViewNode::GetChild(int n) const
{  return (TViewNode*)TTreeNode::GetChild(n);
}

inline TViewNode* TViewNode::GetParent() const
{  return (TViewNode*)Parent;
}

inline TSize&     TViewNode::NodeSize()
{  return Size; }
inline int        TViewNode::NodeWidth() const
{  return Size.cx; }
inline int        TViewNode::NodeHeight() const
{  return Size.cy; }
inline BOOL       TViewNode::GetFolded() const
{  return Folded; }
inline void       TViewNode::SetFolded(BOOL folded)
{  Folded = folded; }
inline void       TViewNode::FlushNodePositions()
{  NodePositions.Flush(); }
const long& max(const long&, const long&);
```

viewnode.cpp

```
//---------------------------------------------------------
---------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TViewNode
//---------------------------------------------------------
---------------------
#include "owlhdr.h"
#pragma hdrstop
#include "tree.h"
#include "mydc.h"
#include "strucdoc.h"
#include "viewnode.h"

TViewNode::TViewNode(int n, int fixed):
   TTerm(n, fixed),Folded(FALSE),
        NodePositions(0,0,2)
{}

void        TViewNode::SetNodeSize(TSize& sz)
{  Size = sz;
}
void        TViewNode::AddNodePosition(TPoint& pt)
{  NodePositions.Add(pt);
}

int         TViewNode::GetNumNodePositions() const
{  return NodePositions.GetItemsInContainer();
}
TPoint&     TViewNode::GetNodePosition(int n) const
{
   return NodePositions[n];
}


TViewNode        *TViewNode::WhichNode(TPoint& point)
{
   TViewNode                      *foundnode = 0;
   TArrayAsVectorIterator<TPoint> iter(NodePositions);

   while(iter && !foundnode) {
      if(TRect(iter++, NodeSize()).Contains(point)) {
         if(!GetFolded()) {
            int numchildren = GetNumChildren();
            for(int i = 0; !foundnode && i<numchildren;
i++)
               foundnode = GetChild(i)-
>WhichNode(point);
         }
         if(!foundnode)
            foundnode = this;
      }
   }
   return foundnode;
}

BOOL        TViewNode::IsVisible()  const
{
   const TViewNode  const *p = this;

   while(p = p->GetParent()) {
      if(p->GetFolded())
         return FALSE;
   }
   return TRUE;
}

void
   TViewNode::FlushChildrenNodePositions()
{
   int          nc = GetNumChildren();

   for(int i = 0; i<nc; i++) {
      GetChild(i)->NodePositions.Flush();

   }
}
```

viewtree.h

```
template<class T>
class   TViewTree: public TTree<T> {
public:
                         TViewTree(TTree<TTerm>
*termtree);
   inline T             *GetFocus() const;
   inline void           SetFocus(T *);
   inline T             *WhichNode(TPoint& point) const;
   void                  CreateMirror(T *parent, int
childnumber, TTerm *term);
private:
   T                    *Focus;
};

template<class T>
TViewTree<T>::TViewTree(TTree<TTerm> *termtree)
{
   if(termtree->GetRoot())
      CreateMirror(0, 0, termtree->GetRoot());
   Focus = GetRoot();
}

template<class T>
void TViewTree<T>::CreateMirror(T *parent, int
childnumber, TTerm *term)
{
   T  *node = CreateOperatorNode((T *)0, term-
>GetOperator());

   if(node) {
      node->SetData(term->GetData(), term-
>GetDataLength());
      if(childnumber < GetNumChildren(parent)) {
         ReplaceChild(parent, childnumber, node);
      } else if(childnumber == GetNumChildren(parent)) {
         AddChild(parent, node);
      }
      int nc = term->GetNumChildren();
      for(int i = 0; i<nc; i++) {
         CreateMirror(node, i, term->GetChild(i));
      }
   }
}

template<class T> inline T
*TViewTree<T>::GetFocus() const
{  return Focus;
}

template<class T> inline void
TViewTree<T>::SetFocus(T *newfocus)
{  Focus = newfocus;
}

template<class T> inline T
*TViewTree<T>::WhichNode(TPoint& point) const
{  return GetRoot()? (T *)GetRoot()->WhichNode(point):0;
}
```

mydc.h

```
#ifndef MYDC_H
#define MYDC_H

const int    RShadW      = 3;
const int    BShadH      = 2;
const int    PlusSize    = 10;
const int    ArrowLength = 9;
const int    ArrowWidth  = 6;

class TMyDC: public TDC {
public:
   void           RectangleOutline(TRect& rect);

   void           ShadowRectangle(TRect& rect, TBrush&
fill, TPen& olpen, TBrush& shadow,
                                TSize& shadowbottleft
= TSize(1, BShadH), TSize& shadowtopright =
TSize(RShadW, 2));
   enum Direction {UP, DOWN, LEFT, RIGHT};
   void           Arrow(TPoint& pos, enum Direction);
   void           Plus(TPoint& pos);
};

#endif
```

mydc.cpp

```
//-------------------------------------------------------
----------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TMyDC
//-------------------------------------------------------
----------------------
#include "owlhdr.h"
#pragma hdrstop
#include "mydc.h"


static const TBrush brush(RGB(255, 255, 0));

void TMyDC::Plus(TPoint& pos)
{
   const int   PlusA = (PlusSize-4)/2;
//   TBrush      brush(PlusColor);
   if(GetClipBox().Touches(TRect(pos, TSize(PlusSize,
PlusSize)))) {
      SelectObject(brush);
      Rectangle(pos.x+PlusA, pos.y, pos.x+(PlusA+3),
pos.y+PlusSize-1);
      Rectangle(pos.x, pos.y+PlusA, pos.x+PlusSize-1,
pos.y+(PlusA+3));
      FillRect(pos.x+PlusA+1, pos.y+PlusA,
pos.x+PlusA+2, pos.y+PlusA+3, brush);
      RestoreBrush();
   }
}
void TMyDC::Arrow(TPoint& pos, enum Direction dir)
{
   TPoint   triangle[3];
   TRect    br;

   triangle[0] = pos;
   switch(dir) {
   case UP:
      br.Set(pos.x-
ArrowWidth/2,pos.y,pos.x+ArrowWidth/2,
pos.y+ArrowLength);
      triangle[1].x = pos.x+ArrowWidth/2; triangle[1].y
= pos.y+ArrowLength;
      triangle[2].x = pos.x-ArrowWidth/2; triangle[2].y
= pos.y+ArrowLength;
      break;
   case DOWN:
      br.Set(pos.x-ArrowWidth/2,pos.y-
ArrowLength,pos.x+ArrowWidth/2, pos.y);
      triangle[1].x = pos.x-ArrowWidth/2; triangle[1].y
= pos.y-ArrowLength;
      triangle[2].x = pos.x+ArrowWidth/2; triangle[2].y
= pos.y-ArrowLength;
      break;
   case LEFT:
      br.Set(pos.x, pos.y-
ArrowWidth/2,pos.x+ArrowLength,pos.y+ArrowWidth/2);
      triangle[1].x = pos.x+ArrowLength;  triangle[1].y
= pos.y-ArrowWidth/2;
      triangle[2].x = pos.x+ArrowLength;  triangle[2].y
= pos.y+ArrowWidth/2;
      break;
   case RIGHT:
      br.Set(pos.x-ArrowLength, pos.y-
ArrowWidth/2,pos.x,pos.y+ArrowWidth/2);
      triangle[1].x = pos.x-ArrowLength;  triangle[1].y
= pos.y+ArrowWidth/2;
      triangle[2].x = pos.x-ArrowLength;  triangle[2].y
= pos.y-ArrowWidth/2;
   }
//   TBrush brush(arrowcolor);
   if(GetClipBox().Touches(br)) {
      SelectObject(brush);
      Polygon(triangle, 3);
      RestoreBrush();
   }
}


void TMyDC::ShadowRectangle(TRect& rect, TBrush& fill,
TPen& pen, TBrush& shadow,
                              TSize& shadowbottleft,
TSize& shadowtopright)
{
   SelectObject(pen);
   SelectObject(fill);
   Rectangle(rect);
   FillRect(rect.left+shadowbottleft.cx, rect.bottom,
            rect.right+shadowtopright.cx,
rect.bottom+shadowbottleft.cy, shadow);
   FillRect(rect.right, rect.top+shadowtopright.cy,
            rect.right+shadowtopright.cx, rect.bottom,
shadow);
   RestoreBrush(); RestorePen();
}

void TMyDC::RectangleOutline(TRect& rect)
{
   MoveTo(rect.TopLeft());
   LineTo(rect.right-1, rect.top);
   LineTo(rect.right-1, rect.bottom-1);
   LineTo(rect.left,    rect.bottom-1);
   LineTo(rect.TopLeft());
}
```

scrwview.h

```
class _DOCVIEWCLASS TScrolledWindowView : public
TWindowView
{
public:
    TScrolledWindowView(TDocument& doc, TWindow *parent
= 0);
   ~TScrolledWindowView() {}
   virtual TSize     CalcSize() = 0;
protected:
    // Message response functions
   void    EvSize(UINT sizeType, TSize&);
   void    AdjustScroller();

   DECLARE_RESPONSE_TABLE(TScrolledWindowView);
};
```

scrwview.cpp

```
//--------------------------------------------------------
----------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TScrolledWindowView
//--------------------------------------------------------
----------------------
#include "owlhdr.h"
#pragma hdrstop

#include "scrwview.h"

DEFINE_RESPONSE_TABLE1(TScrolledWindowView, TWindowView)
   EV_WM_SIZE,
END_RESPONSE_TABLE;

TScrolledWindowView::TScrolledWindowView(TDocument& doc,
TWindow *parent):
               TWindowView(doc, parent)
{
   Attr.Style     |= WS_VSCROLL | WS_HSCROLL;
   Scroller = new TScroller(this, 1, 1, 200, 200);
}

void
TScrolledWindowView::AdjustScroller()
{
   TRect           clientrect;
   GetClientRect(clientrect);

   TSize           sizediff = CalcSize()-
clientrect.Size();

  // only show scrollbars when image is larger than
  // the client area
  //
   Scroller->SetRange(Max(sizediff.cx, 0),
Max(sizediff.cy, 0));
   if (!GetUpdateRect(clientrect, FALSE))
     Invalidate(FALSE);
}

void TScrolledWindowView::EvSize(UINT SizeType, TSize&
Size)
{
   TWindow::EvSize(SizeType, Size);
   if (SizeType != SIZEICONIC) {
      AdjustScroller();
      Invalidate(FALSE);
   }
}
```

scrtrwvw.h

```
//--------------------------------------------------------
----------------------
//   Copyright 1995 Hugo Lyppens
//   Implements class TScrolledTreeWindowView
//--------------------------------------------------------
----------------------




template<class T>
class _DOCVIEWCLASS TScrolledTreeWindowView : public
TScrolledWindowView
{
public:
    TScrolledTreeWindowView(TStructDocument& doc,
TWindow *parent = 0);
    static const char far* StaticName();

    //
    // overridden virtuals from TView
    LPCSTR   GetViewName(){ return StaticName();}

    TSize    CalcSize();

protected:
    virtual  char    GetOpKey( Operator op) { return 0;
}
    virtual  char   *GetOpMenuString( Operator op) {
return ::GetOpName(op); }

    void      Paint(TDC&, BOOL erase, TRect&);
    LRESULT  EvCommand(UINT id, HWND hWndCtl, UINT
notifyCode);

    // Message response functions
    BOOL      VnMirrorSubtree(TNodeAddress *address);
    BOOL      VnGlobalFocus(TNodeAddress *address);
    BOOL      VnNewActive(TNodeAddress *address);

    void      EvSetFocus(HWND);
    void      EvLButtonDown(UINT ModKeys, TPoint&
point);
    void      EvRButtonDown(UINT ModKeys, TPoint&
point);
    void      EvLButtonDblClk(UINT ModKeys, TPoint&
point);
    void      EvChar(UINT key, UINT repeatcount, UINT
flags);
    void      EvKeyDown(UINT key, UINT repeatcount, UINT
flags);
    void      EvKeyUp(UINT key, UINT repeatcount, UINT
flags);

    void      ShowMenu(const TPoint&);
    void      HandleButtonDown(TPoint& point, BOOL
nofold = FALSE);
    void      HandleLButtonDown(TPoint& point);
    void      ToggleFold();

    void      ChangeFocus(T *);
    void      Parent();
    void      PrevNode();
    void      NextNode();
    void      PrevHole();
    void      NextHole();

    void      AddHoleToList();
    void      AddHoleBeforeAfter(WPARAM);
    void      Delete();
    void      Cut();
    void      Copy();
    void      Paste();

    DECLARE_RESPONSE_TABLE(TScrolledTreeWindowView);

    TViewTree<T>     Tree;
    TStructDocument *StructDoc;  // same as Doc member,
but cast to derived class
    BOOL             ShiftDown;
};

NOTIFY_SIG(vnMirrorSubtree, TNodeAddress*);
NOTIFY_SIG(vnGlobalFocus,   TNodeAddress*);
NOTIFY_SIG(vnNewActive,     TNodeAddress*);

#define EV_VN_MIRRORSUBTREE VN_DEFINE(vnMirrorSubtree,
VnMirrorSubtree, pointer)
```

```
#define EV_VN_GLOBALFOCUS   VN_DEFINE(vnGlobalFocus,
VnGlobalFocus,   pointer)
#define EV_VN_NEWACTIVE     VN_DEFINE(vnNewActive,
VnNewActive,     pointer)

template<class T>
void     TScrolledTreeWindowView<T>::EvSetFocus(HWND)
{
   static TNodeAddress   address;

   if(Tree.GetFocus()) {
      Tree.GetNodeAddress(Tree.GetFocus(), address);
      StructDoc->SetActiveAddress(address);
   }
}

template<class T>
void     TScrolledTreeWindowView<T>::EvKeyDown(UINT key,
UINT , UINT )
{
   if(key ==  VK_SHIFT)
      ShiftDown = TRUE;
}


template<class T>
void     TScrolledTreeWindowView<T>::EvKeyUp(UINT key,
UINT , UINT )
{
   if(key ==  VK_SHIFT)
      ShiftDown = FALSE;
}



template<class T>
void     TScrolledTreeWindowView<T>::Delete()
{
   static TNodeAddress address;
   T                  *node = Tree.GetFocus();

   if(node->GetOperator() >= OP_FIRST) {
      Tree.GetNodeAddress(node, address);
      StructDoc->Delete(address);
      Tree.SetFocus(Tree.NodeAt(address));
   } else {
      int cn = Tree.GetMyChildNumber(node);
      node = Tree.GetParent(node);
      if(node && node->IsList()) {
         Tree.GetNodeAddress(node, address);
         StructDoc->ListDelete(address, cn);
         node = Tree.NodeAt(address);
         int numchildren = Tree.GetNumChildren(node);
         if(numchildren>0) {
            if(cn >= numchildren)
               cn  = numchildren-1;
            node = Tree.GetChild(node, cn);
         }
         Tree.SetFocus(node);
         if(node->GetOperator() >= OP_FIRST) {
            NextHole();
         }
      }
   }
}

template<class T>
BOOL
TScrolledTreeWindowView<T>::VnMirrorSubtree(TNodeAddress
*address)
{
   TTerm         *termnode = StructDoc->GetTree()-
>NodeAt(*address);
   T *viewnode = Tree.NodeAt(*address);
   T *curfocus = Tree.GetFocus();
   BOOL setfocus  = (curfocus->IsDescendant(viewnode))
||
                    (curfocus == viewnode);

   Tree.CreateMirror(Tree.GetParent(viewnode),
                     Tree.GetMyChildNumber(viewnode),
                     termnode);
   AdjustScroller();
   if(setfocus)
      Tree.SetFocus(Tree.NodeAt(*address));
   Invalidate();
   return FALSE;
}

template<class T>
BOOL
TScrolledTreeWindowView<T>::VnGlobalFocus(TNodeAddress
*address)
{
   ChangeFocus(Tree.NodeAt(*address));
   return FALSE;
}

template<class T>
BOOL
TScrolledTreeWindowView<T>::VnNewActive(TNodeAddress *)
{
   return FALSE;
}


template<class T>
void      TScrolledTreeWindowView<T>::Cut()
{
    static TNodeAddress address;

    Tree.GetNodeAddress(Tree.GetFocus(), address);
    StructDoc->Cut(address, *this);
    Tree.SetFocus(Tree.NodeAt(address));
}

template<class T>
void      TScrolledTreeWindowView<T>::Copy()
{
    static TNodeAddress address;

    Tree.GetNodeAddress(Tree.GetFocus(), address);
    StructDoc->Copy(address, *this);
}

template<class T>
void      TScrolledTreeWindowView<T>::Paste()
{
    static TNodeAddress address;

    Tree.GetNodeAddress(Tree.GetFocus(), address);
    StructDoc->Paste(address, *this);
    Tree.SetFocus(Tree.NodeAt(address));
}

template<class T>
void      TScrolledTreeWindowView<T>::Parent()
{
   T  *focus    = Tree.GetFocus();
   if(focus = Tree.GetParent(focus))
      ChangeFocus(focus);
}

template<class T>
void      TScrolledTreeWindowView<T>::NextHole()
{
   T  *focus    = Tree.GetFocus();
   T  *nexthole = focus;
   do {
      nexthole = Tree.GetNextNode(nexthole);
      if(!nexthole)
         nexthole = Tree.GetRoot();
   } while (nexthole != focus && (nexthole-
>GetOperator() >= OP_FIRST || !nexthole->IsVisible()));
   if(focus != nexthole)
      ChangeFocus(nexthole);
}

template<class T>
void      TScrolledTreeWindowView<T>::PrevHole()
{
   T  *focus    = Tree.GetFocus();
   T  *prevhole = focus;
   do {
      prevhole = Tree.GetPrevNode(prevhole);
      if(!prevhole)
         prevhole = Tree.GetLastNode();
   } while (prevhole != focus && (prevhole-
>GetOperator() >= OP_FIRST || !prevhole->IsVisible()));
   if(focus != prevhole)
      ChangeFocus(prevhole);
}

template<class T>
void      TScrolledTreeWindowView<T>::NextNode()
{
   T  *focus    = Tree.GetFocus();
   T  *nextnode = focus;

   do {
      nextnode = Tree.GetNextNode(nextnode);
      if(!nextnode)
         nextnode = Tree.GetRoot();
   } while(!nextnode->IsVisible());
   if(focus != nextnode)
      ChangeFocus(nextnode);
}

template<class T>
void      TScrolledTreeWindowView<T>::PrevNode()
{
   T  *focus    = Tree.GetFocus();
   T  *prevnode = focus;

   do {
      prevnode = Tree.GetPrevNode(prevnode);
      if(!prevnode)
         prevnode = Tree.GetLastNode();
   } while(!prevnode->IsVisible());
   if(focus != prevnode)
      ChangeFocus(prevnode);
}

template<class T>
void      TScrolledTreeWindowView<T>::EvChar(UINT key,
UINT repeatcount, UINT flags)
{
   T  *focus = Tree.GetFocus();

   TWindow::EvChar(key, repeatcount, flags);
   switch(key) {
   case 13:
      ShowMenu(focus->GetNodePosition(0)+
        TPoint(focus->NodeWidth()/2, focus-
>NodeHeight()/2)-
        TPoint(Scroller->XPos, Scroller->YPos));
      break;
   default:
       Sort     sort = focus->GetSort();
       Operator op   = focus->GetOperator();
      if(sort == op) {
         for(op = OP_FIRST; op <= OP_LAST; op++) {
            if(GetOpSort(op)==sort && key ==
GetOpKey(op)) {
               EvCommand(CM_REFINE+(int)op, 0, 0);
            }
         }
      }
   }
}


template<class T>
LRESULT  TScrolledTreeWindowView<T>::EvCommand(UINT id,
HWND hWndCtl, UINT notifyCode)
{
      static     TNodeAddress address;
      T *newfocus;

      if(id>=CM_REFINE && id<=CM_REFINE+(int)OP_LAST) {
         Tree.GetNodeAddress(Tree.GetFocus(), address);
         StructDoc->Refine(address, ( Operator)(id-
CM_REFINE), this);
         newfocus = Tree.NodeAt(address);
         if(newfocus->GetNumChildren()) {
            newfocus = Tree.GetChild(newfocus, 0);
         }
         Tree.SetFocus(newfocus);
      }
      return TWindow::EvCommand(id, hWndCtl,
notifyCode);
}

template<class T>
void TScrolledTreeWindowView<T>::AddHoleToList()
{
   static   TNodeAddress     address;
   T     *node = Tree.GetFocus();
   Tree.GetNodeAddress(node, address);
   StructDoc->ListInsert(address, node-
>GetNumChildren());
   Tree.SetFocus(Tree.NodeAt(address));
}

template<class T>
void
TScrolledTreeWindowView<T>::AddHoleBeforeAfter(WPARAM
id)
{
   static   TNodeAddress     address;
   T     *node = Tree.GetFocus();
   int    cn   = Tree.GetMyChildNumber(node);
   while((node = Tree.GetParent(node)) && !node-
>IsList()) {
      cn = Tree.GetMyChildNumber(node);
   }
   if(node) {
      if(id == CM_ADDHOLEAFTER)
         cn++;
      Tree.GetNodeAddress(node, address);
      StructDoc->ListInsert(address, cn);
      node = Tree.NodeAt(address);
      Tree.SetFocus(Tree.GetChild(node, cn));
   }
}

template<class T>
TScrolledTreeWindowView<T>::TScrolledTreeWindowView(TStr
uctDocument& doc, TWindow *parent):
              TScrolledWindowView(doc, parent),
StructDoc(&doc), Tree(doc.GetTree()),
              ShiftDown(FALSE)
{
   static TNodeAddress   address;

   StructDoc->GetActiveAddress(address);
   Tree.SetFocus(Tree.NodeAt(address));
   Attr.AccelTable = IDA_FLOWVIEW;
   SetViewMenu(new
TMenuDescr(IDM_FLOWVIEW,0,1,0,0,0,1));
   SetBkgndColor(TColor::LtGray);
}

template<class T>
void TScrolledTreeWindowView<T>::EvRButtonDown(UINT
ModKeys, TPoint& point)
{
   TWindow::EvRButtonDown(ModKeys, point);


   TPoint          pt(Scroller->XPos, Scroller->YPos);
   T  *newfocus = Tree.WhichNode(point + pt);

   if(newfocus) {
      if(newfocus != Tree.GetFocus())
         ChangeFocus(newfocus);
   }
   ShowMenu(point);
}

template<class T>
void TScrolledTreeWindowView<T>::ShowMenu(const TPoint&
point)
{
      TPoint         p    = point;
      TPopupMenu     m;
      T *node = Tree.GetFocus();

      Sort     sort = node->GetSort();
      Operator op   = node->GetOperator();
      char  s[50], *opname;

      if(sort == op) {
         for(op = OP_FIRST; op<=OP_LAST;
             op++) {
            if(GetOpSort(op)==sort) {
               m.AppendMenu(MF_ENABLED,
CM_REFINE+(int)op, GetOpMenuString(op));
            }
         }
         m.AppendMenu(MF_SEPARATOR, 0, 0);
         m.AppendMenu(StructDoc->ClipboardOK(sort,
*this)?
                        MF_ENABLED:MF_GRAYED,
                   CM_PASTE,  "&Paste\tCtrl+V");

         T  *par = Tree.GetParent(node);
         if(par && par->IsList()) {
            m.AppendMenu(MF_SEPARATOR, 0, 0);
            wsprintf(s, "Delete from %s\tDelete",
GetOpName(par->GetOperator()));
            m.AppendMenu(MF_ENABLED, CM_DELETE, s);
         }
      } else {
         m.AppendMenu(MF_ENABLED, CM_FOLD, node-
>GetFolded()?"Unfold\tF":"Fold\tF");
         m.AppendMenu(MF_SEPARATOR, 0, 0);
         m.AppendMenu(MF_ENABLED, CM_CUT,
"Cu&t\tCtrl+X");
         m.AppendMenu(MF_ENABLED, CM_COPY,
"&Copy\tCtrl+C");
         m.AppendMenu(MF_ENABLED, CM_DELETE,
"&Delete\tDelete");
```

```
      }
      if(node->IsList()) {
         m.AppendMenu(MF_SEPARATOR, 0, 0);
         opname = GetOpName(node->GetOperator());
         wsprintf(s, "Add Hole to %s\tH", opname);
         m.AppendMenu(MF_ENABLED, CM_ADDHOLETOLIST,  s);
      }

      T  *par = node;
      while(par = Tree.GetParent(par)) {
         if(par->IsList()) {
            m.AppendMenu(MF_SEPARATOR, 0, 0);
            opname = GetOpName(par->GetOperator());
            wsprintf(s, "Add to ancestor %s Before\tB",
opname);
            m.AppendMenu(MF_ENABLED, CM_ADDHOLEBEFORE,
s);
            wsprintf(s, "Add to ancestor %s After\tA",
opname);
            m.AppendMenu(MF_ENABLED, CM_ADDHOLEAFTER,
s);
            break;
         }
      }
      m.AppendMenu(MF_SEPARATOR, 0, 0);
      m.AppendMenu(MF_ENABLED, CM_NEXTHOLE,  "Next
Hole\tDown");
      m.AppendMenu(MF_ENABLED, CM_PREVHOLE,  "Prev
Hole\tUp");
      m.AppendMenu(MF_ENABLED, CM_NEXTNODE,  "Next
Node\tRight");
      m.AppendMenu(MF_ENABLED, CM_PREVNODE,  "Prev
Node\tLeft");
      m.AppendMenu(MF_ENABLED, CM_PARENT,
"Parent\tP");

      ClientToScreen(p);
      m.TrackPopupMenu(TPM_RIGHTBUTTON|TPM_LEFTALIGN,
p,0,(HWND)*this);
}

template<class T>
void TScrolledTreeWindowView<T>::EvLButtonDblClk(UINT
ModKeys, TPoint& point)
{
   TWindow::EvLButtonDblClk(ModKeys, point);
   HandleLButtonDown(point);
}

template<class T>
void TScrolledTreeWindowView<T>::EvLButtonDown(UINT
ModKeys, TPoint& point)
{
   TWindow::EvLButtonDown(ModKeys, point);
   HandleLButtonDown(point);
}

template<class T>
void
TScrolledTreeWindowView<T>::HandleLButtonDown(TPoint&
point)
{
   static   TNodeAddress     address;

   HandleButtonDown(point, ShiftDown);
   if(ShiftDown) {
      Tree.GetNodeAddress(Tree.GetFocus(), address);
      StructDoc->GlobalFocus(address);
   }
}

template<class T>
void TScrolledTreeWindowView<T>::ChangeFocus(T
*newfocus)
{
   T  *oldfocus = Tree.GetFocus();
   static   TNodeAddress     address;

   while(newfocus && !newfocus->IsVisible())
      newfocus = Tree.GetParent(newfocus);

   if(newfocus && newfocus!=oldfocus) {
      TPoint          pt(Scroller->XPos, Scroller-
>YPos);
      Tree.SetFocus(newfocus);
      int np, i;
      np = oldfocus->GetNumNodePositions();
      for(i = 0; i<np; i++) {
         InvalidateRect(TRect(oldfocus-
>GetNodePosition(i)-pt,
                              oldfocus->NodeSize()));
      }
      np = newfocus->GetNumNodePositions();
      for(i = 0; i<np; i++) {
         InvalidateRect(TRect(newfocus-
>GetNodePosition(i)-pt,
                              newfocus->NodeSize()));
      }
      Tree.GetNodeAddress(newfocus, address);
      StructDoc->SetActiveAddress(address);
   }
}

template<class T>
void TScrolledTreeWindowView<T>::ToggleFold()
{
   T  *focus = Tree.GetFocus();

   if(Tree.GetNumChildren(focus)>0) {
      focus->SetFolded(!focus->GetFolded());
      AdjustScroller();
      Invalidate();
   }
}

template<class T>
void
TScrolledTreeWindowView<T>::HandleButtonDown(TPoint&
point, BOOL nofold)
{
      TPoint  pt(Scroller->XPos, Scroller->YPos);
      T  *newfocus = Tree.WhichNode(point + pt);
      T  *oldfocus = Tree.GetFocus();

      if(newfocus) {
         if(newfocus != oldfocus) {
            ChangeFocus(newfocus);
         } else if(!nofold) {
            ToggleFold();
         }
      }
}

template<class T>
TSize    TScrolledTreeWindowView<T>::CalcSize()
{
   T   *root = Tree.GetRoot();

   if(root) {
      root->CalcSize((TMyDC&)TClientDC(HWindow));
      return root->NodeSize();
   }
   return TSize(0, 0);
}
```

owlhdr.h

```
#include <string.h>
#include <classlib\arrays.h>

#include <owl\dc.h>
#include <owl\scroller.h>
#include <owl\statusba.h>
#include <owl\editview.h>
#include <owl\listview.h>
#include <owl\docmanag.h>
#include <owl\filedoc.h>
#include <owl\dialog.h>
#include <owl\inputdia.h>
#include <owl\applicat.h>
#include <owl\mdi.h>
#include <owl\decmdifr.h>
#include <owl\color.h>
#include <owl\owlpch.h>
```

B PL4 signature and view modules

File	Description
sign_pl4.h	• PL4 signature module header
sign_pl4.cpp	• PL4 signature module source
list_pl4.cpp	• implements listing view
flow_pl4.cpp	• implements flow diagram view
ctxt_pl4.cpp	• implements active context view
euclides.str	• PL4 euclides example program file representation

sign_pl4.h

```
enum Operator {
   OP_NONE,
/* HOLES */
   OP_PROGHOLE, OP_DECHOLE, OP_NDECHOLE, OP_DECSHOLE,
   OP_IDHOLE,   OP_TYPEHOLE, OP_STATHOLE, OP_STATSHOLE,
OP_VARSHOLE, OP_EXPRSHOLE,OP_EXPRHOLE,
   OP_ALTS1HOLE,OP_ALTS2HOLE, OP_ALT2HOLE,
OP_LABELSHOLE, OP_LABELHOLE,
   OP_PBODYHOLE,OP_PPARSHOLE,
   OP_PPARHOLE, OP_PARGSHOLE, OP_PARGHOLE, OP_FBODYHOLE,
OP_FPARSHOLE,
   OP_FPARHOLE, OP_FARGSHOLE,
/* Real operators */
   OP_PROG, OP_DECS, OP_NDECS, OP_NDEC, OP_PROCDEC,
OP_FUNCDEC,
   OP_ID, OP_INTTYPE, OP_BOOLTYPE, OP_CHARTYPE,
   OP_REALTYPE,OP_STATS, OP_SKIP,
   OP_CONCASSIGN, OP_PROCCALL, OP_PROCCALLARGS,
OP_IFTHEN, OP_IFTHENELSE,
   OP_WHILE, OP_READ, OP_WRITE, OP_FOR, OP_BLOCK,
OP_VARS, OP_EXPRS,
   OP_VARVALUE, OP_FUNCCALL, OP_FUNCCALLARGS,
OP_NUMVALUE, OP_CHARVALUE,
   OP_IMPLICATION, OP_EQUIVALENCE, OP_OR,  OP_AND,
OP_LESS,  OP_LESSEQ,
   OP_GREATER,  OP_GREATEREQ, OP_EQUALITY,
OP_DIFFERENCE, OP_ADDITION,
   OP_SUBTRACTION, OP_MULTIPLICATION, OP_DIVISION,
OP_MODULO,
   OP_UNARYMINUS, OP_NOT, OP_EXPRBLOCK, OP_PBODYSTATS,
OP_PBODYPARSSTATS,
   OP_PPARS, OP_PPARVALUE, OP_PPARREF, OP_PARGS,
OP_PARGVALUE, OP_PARGREF,
   OP_FBODY, OP_FPARS, OP_FPARVALUE, OP_FARGS,
   OP_NUMCASE, OP_LABCASE, OP_ALTS1, OP_ALTS2, OP_ALT2,
OP_LABELS,
   OP_CHARLABEL, OP_NUMLABEL
};


DEFINE_SORT_AND_HOLE(PROG)
DEFINE_SORT_AND_HOLE(DEC)
DEFINE_SORT_AND_HOLE(NDEC)
DEFINE_SORT_AND_HOLE(DECS)
DEFINE_SORT_AND_HOLE(ID)
DEFINE_SORT_AND_HOLE(TYPE)
DEFINE_SORT_AND_HOLE(STAT)
DEFINE_SORT_AND_HOLE(STATS)
DEFINE_SORT_AND_HOLE(VARS)
DEFINE_SORT_AND_HOLE(EXPRS)
DEFINE_SORT_AND_HOLE(EXPR)
DEFINE_SORT_AND_HOLE(ALTS1)
DEFINE_SORT_AND_HOLE(ALTS2)
DEFINE_SORT_AND_HOLE(ALT2)
DEFINE_SORT_AND_HOLE(LABELS)
DEFINE_SORT_AND_HOLE(LABEL)
DEFINE_SORT_AND_HOLE(PBODY)
DEFINE_SORT_AND_HOLE(PPARS)
DEFINE_SORT_AND_HOLE(PPAR)
DEFINE_SORT_AND_HOLE(PARGS)
DEFINE_SORT_AND_HOLE(PARG)
DEFINE_SORT_AND_HOLE(FBODY)
DEFINE_SORT_AND_HOLE(FPARS)
DEFINE_SORT_AND_HOLE(FPAR)
DEFINE_SORT_AND_HOLE(FARGS)
DEFINE_OP2(PROG,            PROG, Declarations, DECS,
Statements, STATS)
DEFINE_OP_LIST(DECS,     DECS,    Declaration, DEC)
DEFINE_OP_LIST(NDECS,       DEC,      VarDeclaration,
NDEC)
DEFINE_OP2(NDEC,            NDEC,     Variable, ID, Type,
TYPE)
DEFINE_OP2(PROCDEC,         DEC,      ProcName, ID, Body,
PBODY)
DEFINE_OP2(FUNCDEC,         DEC,      FuncName, ID, Body,
FBODY)
DEFINE_OP0(ID,              ID)
DEFINE_OP0(INTTYPE,         TYPE)
DEFINE_OP0(BOOLTYPE,        TYPE)
DEFINE_OP0(CHARTYPE,        TYPE)
DEFINE_OP0(REALTYPE,        TYPE)
DEFINE_OP_LIST(STATS,    STATS,Statement, STAT)
DEFINE_OP0(SKIP,            STAT)
DEFINE_OP2(CONCASSIGN,      STAT,    Variables, VARS,
Expressions, EXPRS)
DEFINE_OP1(PROCCALL,        STAT,    ProcName, ID)
DEFINE_OP2(PROCCALLARGS, STAT,    ProcName, ID,
Arguments, PARGS)
DEFINE_OP2(IFTHEN,      STAT,    Guard, EXPR,
TrueStatements, STATS)
DEFINE_OP3(IFTHENELSE,      STAT,    Guard, EXPR,
TrueStatements, STATS, FalseStatements, STATS)
DEFINE_OP2(WHILE,           STAT,    Guard, EXPR,
Statements, STATS)
```

```
/*DEFINE_OP3(     45, NUMCASE,      STAT,
   Expression, EXPR, Alternatives, ALTS1, OutStatements,
STATS)
DEFINE_OP3(    46, LABCASE,      STAT,    Expression,
EXPR, LabAlternatives, ALTS2, OutStatements, STATS)*/
DEFINE_OP_LIST(READ,       STAT,    Variable, ID)
DEFINE_OP_LIST(WRITE,   STAT,    Expression, EXPR)
DEFINE_OP5(FOR,         STAT,    Variable, ID,
FromExpression, EXPR, ByExpression, EXPR, ToExpression,
EXPR, Statements, STATS)
DEFINE_OP2(BLOCK,          STAT,    Declarations, DECS,
Statements, STATS)
DEFINE_OP_LIST(VARS,       VARS,    Variable, ID)
DEFINE_OP_LIST(EXPRS,   EXPRS,Expression, EXPR)
DEFINE_OP1(VARVALUE,       EXPR,    Variable, ID)
DEFINE_OP1(FUNCCALL,       EXPR,    FuncName, ID)
DEFINE_OP2(FUNCCALLARGS,EXPR,    FuncName, ID,
Arguments, FARGS)
DEFINE_OP0(NUMVALUE,       EXPR)
DEFINE_OP0(CHARVALUE,   EXPR)
DEFINE_OP2(IMPLICATION,    EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(EQUIVALENCE,    EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(OR,             EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(AND,         EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(LESS,           EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(LESSEQ,      EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(GREATER,        EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(GREATEREQ,   EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(EQUALITY,       EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(DIFFERENCE,     EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(ADDITION,       EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(SUBTRACTION, EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(MULTIPLICATION, EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(DIVISION,       EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP2(MODULO,      EXPR,    Arg0, EXPR, Arg1,
EXPR)
DEFINE_OP1(UNARYMINUS,     EXPR,    Arg0, EXPR)
DEFINE_OP1(NOT,         EXPR,    Arg0, EXPR)
DEFINE_OP3(EXPRBLOCK,   EXPR,    Declarations, DECS,
Statements, STATS, Expression, EXPR)
DEFINE_OP_LIST(PBODYSTATS, PBODY,Statement, STAT)
DEFINE_OP2(PBODYPARSSTATS, PBODY,ProcParameters, PPARS,
Statements, STATS)
DEFINE_OP_LIST(PPARS,   PPARS,ProcParameter, PPAR)
DEFINE_OP2(PPARVALUE,   PPAR,    Variable, ID, Type,
TYPE)
DEFINE_OP2(PPARREF,        PPAR,    Variable, ID, Type,
TYPE)
DEFINE_OP_LIST(PARGS,   PARGS,ProcArgument, PARG)
DEFINE_OP1(PARGVALUE,   PARG,    Expression, EXPR)
DEFINE_OP1(PARGREF,        PARG,    Variable, ID)
DEFINE_OP2(FBODY,          FBODY,FuncParameters, FPARS,
Expression, EXPR)
DEFINE_OP_LIST(FPARS,   FPARS,FuncParameter, FPAR)
DEFINE_OP2(FPARVALUE,   FPAR,    Variable, ID, Type,
TYPE)
DEFINE_OP_LIST(FARGS,   FARGS,FuncArgument, EXPR)
DEFINE_OP3(NUMCASE,        STAT,    Case, EXPR, In,
ALTS1, Out, STATS)
DEFINE_OP3(LABCASE,        STAT,    Case, EXPR, In,
ALTS2, Out, STATS)
DEFINE_OP_LIST(ALTS1,      ALTS1,Statements, STATS)
DEFINE_OP_LIST(ALTS2,   ALTS2,Alternative, ALT2)
DEFINE_OP2(ALT2,           ALT2,    Labels, LABELS,
Statements, STATS)
DEFINE_OP_LIST(LABELS,     LABELS,  Label, LABEL)
DEFINE_OP0(NUMLABEL,    LABEL)
DEFINE_OP0(CHARLABEL,      LABEL)

const Operator OP_FIRSTHOLE = OP_PROGHOLE;
const Operator OP_LASTHOLE  = OP_FARGSHOLE;
const Operator OP_FIRST     = OP_PROG;
const Operator OP_LAST      = OP_CHARLABEL;

const Sort     TERMSORT     = SORT_PROG;

template<class T>
```

```
T              *CreateOperatorNode(T *,const Operator
op)
{
   T              *node = 0;

   switch(op) {
   case OP_PROGHOLE:node = new TPROGHOLE<T>; break;
   case OP_DECHOLE: node = new TDECHOLE<T>; break;
   case OP_NDECHOLE: node = new TNDECHOLE<T>; break;
   case OP_DECSHOLE: node = new TDECSHOLE<T>; break;
   case OP_IDHOLE:node = new TIDHOLE<T>; break;
   case OP_TYPEHOLE: node = new TTYPEHOLE<T>; break;
   case OP_STATHOLE: node = new TSTATHOLE<T>; break;
   case OP_STATSHOLE:node = new TSTATSHOLE<T>; break;
   case OP_VARSHOLE: node = new TVARSHOLE<T>; break;
   case OP_EXPRSHOLE:node = new TEXPRSHOLE<T>; break;
   case OP_EXPRHOLE: node = new TEXPRHOLE<T>; break;
   case OP_ALTS1HOLE:node = new TALTS1HOLE<T>; break;
   case OP_ALTS2HOLE:node = new TALTS2HOLE<T>; break;
   case OP_ALT2HOLE: node = new TALT2HOLE<T>; break;
   case OP_LABELSHOLE:node = new TLABELSHOLE<T>; break;
   case OP_LABELHOLE:node = new TLABELHOLE<T>; break;
   case OP_PBODYHOLE:node = new TPBODYHOLE<T>; break;
   case OP_PPARSHOLE:node = new TPPARSHOLE<T>; break;
   case OP_PPARHOLE: node = new TPPARHOLE<T>; break;
   case OP_PARGSHOLE:node = new TPARGSHOLE<T>; break;
   case OP_PARGHOLE: node = new TPARGHOLE<T>; break;
   case OP_FBODYHOLE:node = new TFBODYHOLE<T>; break;
   case OP_FPARSHOLE:node = new TFPARSHOLE<T>; break;
   case OP_FPARHOLE: node = new TFPARHOLE<T>; break;
   case OP_FARGSHOLE:node = new TFARGSHOLE<T>; break;

   case OP_PROG:     node = new TPROG<T>; break;
   case OP_BLOCK:    node = new TBLOCK<T>; break;
   case OP_DECS:     node = new TDECS<T>; break;
   case OP_NDECS:    node = new TNDECS<T>; break;
   case OP_NDEC:     node = new TNDEC<T>; break;
   case OP_PROCDEC:  node = new TPROCDEC<T>; break;
   case OP_FUNCDEC:  node = new TFUNCDEC<T>; break;
   case OP_ID:       node = new TID<T>; break;
   case OP_INTTYPE:  node = new TINTTYPE<T>; break;
   case OP_BOOLTYPE: node = new TBOOLTYPE<T>; break;
   case OP_CHARTYPE: node = new TCHARTYPE<T>; break;
   case OP_REALTYPE: node = new TREALTYPE<T>; break;
   case OP_STATS:    node = new TSTATS<T>; break;
   case OP_SKIP:     node = new TSKIP<T>; break;
   case OP_CONCASSIGN:node = new TCONCASSIGN<T>; break;
   case OP_PROCCALL: node = new TPROCCALL<T>; break;
   case OP_PROCCALLARGS: node = new TPROCCALLARGS<T>;
break;
   case OP_IFTHEN:node = new TIFTHEN<T>; break;
   case OP_IFTHENELSE:node = new TIFTHENELSE<T>; break;
   case OP_WHILE:    node = new TWHILE<T>; break;
   case OP_READ:     node = new TREAD<T>; break;
   case OP_WRITE:    node = new TWRITE<T>; break;
   case OP_FOR:   node = new TFOR<T>; break;
   case OP_VARS:     node = new TVARS<T>; break;
   case OP_EXPRS:    node = new TEXPRS<T>; break;
   case OP_VARVALUE: node = new TVARVALUE<T>; break;
   case OP_FUNCCALL: node = new TFUNCCALL<T>; break;
   case OP_FUNCCALLARGS: node = new TFUNCCALLARGS<T>;
break;
   case OP_NUMVALUE: node = new TNUMVALUE<T>; break;
   case OP_CHARVALUE:node = new TCHARVALUE<T>; break;
   case OP_IMPLICATION:node = new TIMPLICATION<T>;
break;
   case OP_EQUIVALENCE:node = new TEQUIVALENCE<T>;
break;
   case OP_OR:       node = new TOR<T>; break;
   case OP_AND:   node = new TAND<T>; break;
   case OP_LESS:     node = new TLESS<T>; break;
   case OP_LESSEQ:node = new TLESSEQ<T>; break;
   case OP_GREATER:  node = new TGREATER<T>; break;
   case OP_GREATEREQ:node = new TGREATEREQ<T>; break;
   case OP_EQUALITY: node = new TEQUALITY<T>; break;
   case OP_DIFFERENCE:node = new TDIFFERENCE<T>; break;
   case OP_ADDITION: node = new TADDITION<T>; break;
   case OP_SUBTRACTION:node = new TSUBTRACTION<T>;
break;
   case OP_MULTIPLICATION:  node = new
TMULTIPLICATION<T>; break;
   case OP_DIVISION: node = new TDIVISION<T>; break;
   case OP_MODULO:node = new TMODULO<T>; break;
   case OP_UNARYMINUS:node = new TUNARYMINUS<T>; break;
   case OP_NOT:   node = new TNOT<T>; break;
   case OP_EXPRBLOCK:node = new TEXPRBLOCK<T>; break;
   case OP_PBODYSTATS:node = new TPBODYSTATS<T>; break;
   case OP_PBODYPARSSTATS:  node = new
TPBODYPARSSTATS<T>; break;
   case OP_PPARS:    node = new TPPARS<T>; break;
   case OP_PPARVALUE:node = new TPPARVALUE<T>; break;
   case OP_PPARREF:   node = new TPPARREF<T>; break;
   case OP_PARGS:     node = new TPARGS<T>; break;
   case OP_PARGVALUE:node = new TPARGVALUE<T>; break;
   case OP_PARGREF:   node = new TPARGREF<T>; break;
   case OP_FBODY:     node = new TFBODY<T>; break;
   case OP_FPARS:     node = new TFPARS<T>; break;
   case OP_FPARVALUE:node = new TFPARVALUE<T>; break;
   case OP_FARGS:     node = new TFARGS<T>; break;
   case OP_NUMCASE:   node = new TNUMCASE<T>; break;
   case OP_LABCASE:   node = new TLABCASE<T>; break;
   case OP_ALTS1:     node = new TALTS1<T>; break;
   case OP_ALTS2:     node = new TALTS2<T>; break;
   case OP_ALT2:      node = new TALT2<T>; break;
   case OP_LABELS: node = new TLABELS<T>; break;
   case OP_CHARLABEL:node = new TCHARLABEL<T>; break;
   case OP_NUMLABEL: node = new TNUMLABEL<T>; break;
   }
   return node;
}
```

sign_pl4.cpp

```
//-----------------------------------------------------
---------------------
//    Copyright 1995 Hugo Lyppens
//    Implements PL4 signature
//-----------------------------------------------------
---------------------
#include "owlhdr.h"
#pragma hdrstop

#include "tree.h"
#include "strucdoc.h"


void     AskData(TWindow *window, TTerm *node)
{
   char     s[MAXOPDATALEN+1], v[20];
   char     h[MAXOPNAMELEN+51];

   s[0] = 0;
   wsprintf(h, "Operator %s additional info", node-
>GetOpName());
   switch(node->GetOperator()) {
   case OP_NUMVALUE:
   case OP_NUMLABEL:
      TInputDialog(window, h, "Enter numeric value:",
                   s, sizeof s).Execute();
      ostrstream(v, sizeof v)<<atof(s)<<'\0';
      node->SetData(v, strlen(v)+1);
      break;
   case OP_CHARVALUE:
   case OP_CHARLABEL:
      TInputDialog(window, h, "Enter a character:",
                   s, sizeof s).Execute();
      s[1] = s[0];
      s[0] = '\'';
      s[2] = '\'';
      s[3] = 0;
      node->SetData(s, 4);
      break;
   case OP_ID:
      TInputDialog(window, h, "Enter an identifier
name",
                   s, sizeof s).Execute();
      node->SetData(s, strlen(s)+1);
      break;
   }
}

Sort      GetOpSort(Operator op)
{
   if(op<OP_FIRST)
      return (Sort)op;
   switch(op) {
   case OP_PROG:            return SORT_PROG;

   case OP_DECS:            return SORT_DECS;

   case OP_PROCDEC:
   case OP_FUNCDEC:
   case OP_NDECS:       return SORT_DEC;

   case OP_NDEC:            return SORT_NDEC;

   case OP_ID:              return SORT_ID;

   case OP_INTTYPE:
   case OP_BOOLTYPE:
   case OP_CHARTYPE:
   case OP_REALTYPE:        return SORT_TYPE;

   case OP_STATS:           return SORT_STATS;
   case OP_SKIP:
   case OP_CONCASSIGN:
   case OP_PROCCALL:
   case OP_PROCCALLARGS:
   case OP_IFTHEN:
   case OP_IFTHENELSE:
   case OP_WHILE:
   case OP_READ:
   case OP_WRITE:
   case OP_FOR:
   case OP_BLOCK:
   case OP_NUMCASE:
   case OP_LABCASE:         return SORT_STAT;

   case OP_VARS:            return SORT_VARS;

   case OP_EXPRS:           return SORT_EXPRS;

   case OP_VARVALUE:
   case OP_FUNCCALL:
   case OP_FUNCCALLARGS:
   case OP_NUMVALUE:
   case OP_CHARVALUE:
   case OP_IMPLICATION:
   case OP_EQUIVALENCE:
   case OP_OR:
   case OP_AND:
   case OP_LESS:
   case OP_LESSEQ:
   case OP_GREATER:
   case OP_GREATEREQ:
   case OP_EQUALITY:
   case OP_DIFFERENCE:
   case OP_ADDITION:
   case OP_SUBTRACTION:
   case OP_MULTIPLICATION:
   case OP_DIVISION:
   case OP_MODULO:
   case OP_UNARYMINUS:
   case OP_NOT:
   case OP_EXPRBLOCK:       return SORT_EXPR;

   case OP_PBODYSTATS:
   case OP_PBODYPARSSTATS:     return SORT_PBODY;

   case OP_PPARS:              return SORT_PPARS;

   case OP_PPARVALUE:
   case OP_PPARREF:            return SORT_PPAR;

   case OP_PARGS:              return SORT_PARGS;

   case OP_PARGVALUE:
   case OP_PARGREF:            return SORT_PARG;

   case OP_FBODY:              return SORT_FBODY;

   case OP_FPARS:              return SORT_FPARS;

   case OP_FPARVALUE:       return SORT_FPAR;

   case OP_FARGS:              return SORT_FARGS;

   case OP_ALTS1:              return SORT_ALTS1;

   case OP_ALTS2:              return SORT_ALTS2;

   case OP_ALT2:               return SORT_ALT2;

   case OP_LABELS:          return SORT_LABELS;

   case OP_CHARLABEL:
   case OP_NUMLABEL:           return SORT_LABEL;
   }
   return SORT_NONE;
}



char     *OpName[] = {
   "PROG", "DECS", "NDECS", "NDEC", "PROCDEC",
"FUNCDEC",
   "ID", "INT", "BOOL", "CHAR",   "REAL",   "STATS",
"SKIP",
   "CONCASSIGN", "PROCCALL", "PROCCALLARGS", "IFTHEN",
"IFTHENELSE",
   "WHILE", "READ", "WRITE", "FOR", "BLOCK", "VARS",
"EXPRS",
   "VARVALUE", "FUNCCALL", "FUNCCALLARGS", "NUMVALUE",
"CHARVALUE",
   "IMPLICATION", "EQUIVALENCE", "OR",  "AND",  "LESS",
"LESSEQ",
   "GREATER",  "GREATEREQ", "EQUALITY", "DIFFERENCE",
"ADDITION",
   "SUBTRACTION", "MULTIPLICATION", "DIVISION",
"MODULO",
   "UNARYMINUS", "NOT", "EXPRBLOCK", "PBODYSTATS",
"PBODYPARSSTATS",
   "PPARS", "PPARVALUE", "PPARREF", "PARGS",
"PARGVALUE", "PARGREF",
   "FBODY", "FPARS", "FPARVALUE", "FARGS",
   "NUMCASE", "LABCASE", "ALTS1", "ALTS2", "ALT2",
   "LABELS", "NUMLABEL", "CHARLABEL"
};

char     *SortName[] = {
   "",
   "PROG", "DEC", "NDEC", "DECS",
   "ID",  "TYPE", "STAT", "STATS", "VARS", "EXPRS",
"EXPR",
   "ALTS1","ALTS2", "ALT2", "LABELS", "LABEL",
   "PBODY","PPARS",
   "PPAR", "PARGS", "PARG", "FBODY", "FPARS",
   "FPAR", "FARGS"
};

char   *GetSortName(Sort sort)
{
   return SortName[(int)sort];
}

char   *GetOpName(Operator op)
{
   if(!op)
      return 0;
   if(op<OP_FIRST) {
               static char     nbuf[MAXOPNAMELEN+1];
      wsprintf(nbuf, "%sHOLE", GetSortName((Sort)op));
      return nbuf;
   } else {
      return OpName[(int)(op-OP_FIRST)];
   }
}
```

list_pl4.cpp

```
//---------------------------------------------------
---------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TListingView
//---------------------------------------------------
---------------------
#include "owlhdr.h"
#pragma hdrstop

//#pragma option -Jgx

#include <owl\menu.h>

#include "views.rc"
#include "tree.h"
#include "strucdoc.h"
#include "mydc.h"
#include "viewnode.h"
#include "viewtree.h"
#include "scrwview.h"
#include "scrtrwvw.h"



class   TListingViewNode : public TViewNode {
public:
                         TListingViewNode(int n = 0,
int fixed = FALSE);
   inline TListingViewNode *GetChild(int n) const;
   void                 CalcSizeFolded(TMyDC&);
   void                 DrawFolded(TMyDC&, TPoint&
pos);
   void                  CalcSizeString(TMyDC&, const
char const *);
   void                  DrawString(TMyDC&, TPoint&,
const char const *);
   void                  CalcSizeHole(TMyDC&);
   void                  DrawHole(TMyDC&, TPoint&);
   void                  DoPosFocusFolded(TMyDC& dc,
TPoint& pos);
   void               CalcSizeDelimiters(TMyDC& dc, \
                                          char *start,
char *between, char *end);

   void               DrawDelimiters(TMyDC& dc, TPoint&
pos, \
                                          char *start,
char *between, char *end);
   static TViewTree<TListingViewNode>     *ListingTree;
};

inline TListingViewNode* TListingViewNode::GetChild(int
n) const
{  return (TListingViewNode*)TTreeNode::GetChild(n);
}


const int         Indent  = 20;
const int         DistX   = 4;
#define  FocusColor   TColor(128, 128, 128)
static TBrush            FocusBrush  =
TBrush(FocusColor);

TViewTree<TListingViewNode>
*TListingViewNode::ListingTree = 0;


TListingViewNode::TListingViewNode(int n, int fixed):
   TViewNode(n, fixed)
{
}
void
TListingViewNode::CalcSizeString(TMyDC& dc, const char
const *s)
{
   if(GetFolded())
      CalcSizeFolded(dc);
   else
      SetNodeSize(dc.GetTextExtent(s, strlen(s)));
}

void
TListingViewNode::DrawString(TMyDC& dc, TPoint& pos,
const char const *s)
{
   DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      dc.TextOut(pos, s);
   }
}
void
   TListingViewNode::DoPosFocusFolded(TMyDC& dc, TPoint&
pos)
{
   AddNodePosition(pos);
   if(ListingTree->GetFocus() == this)
      dc.FillRect(TRect(pos, NodeSize()), FocusBrush);
   if(GetFolded())
      DrawFolded(dc, pos);
   FlushChildrenNodePositions();
}
void
   TListingViewNode::CalcSizeFolded(TMyDC& dc)
{
   char  str[80];

   wsprintf(str, "%s <%s>", GetOpName(), GetSortName());
   SetNodeSize(dc.GetTextExtent(str,
strlen(str))+TSize(PlusSize+DistX, 0));
}

void              TListingViewNode::DrawFolded(TMyDC&
dc, TPoint& pos)
{
   char  str[80];

   wsprintf(str, "%s <%s>", GetOpName(), GetSortName());
   dc.Plus(pos+TSize(0, (NodeSize().cy-PlusSize)/2));
   dc.TextOut(pos.x+PlusSize+DistX, pos.y, str);
}


void
   TListingViewNode::CalcSizeHole(TMyDC& dc)
{
   static char    s[40] = "? <";
   char  *p = GetSortName();
   strcpy(s+3, p); strcpy(s+3+strlen(p), ">");
   CalcSizeString(dc, s);
}

void                 TListingViewNode::DrawHole(TMyDC&
dc, TPoint& pos)
{
   static char    s[40] = "? <";
   char  *p = GetSortName();
   strcpy(s+3, p); strcpy(s+3+strlen(p), ">");
   DrawString(dc, pos, s);
}

void  TPROGHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPROGHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TDECHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TDECHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TDECSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TDECSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TNDECHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TNDECHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TIDHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TIDHOLE<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TTYPEHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TTYPEHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TSTATHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TSTATHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TSTATSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TSTATSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TVARSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TVARSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TEXPRSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TEXPRSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TEXPRHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TEXPRHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TALTS1HOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALTS1HOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TALTS2HOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALTS2HOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TALT2HOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALT2HOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TLABELSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TLABELSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TLABELHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TLABELHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPBODYHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPBODYHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPPARSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPPARSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPPARHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPPARHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPARGSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPARGSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPARGHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPARGHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TFBODYHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFBODYHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TFPARSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFPARSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TFPARHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFPARHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TFARGSHOLE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFARGSHOLE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }


void  TPROG<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortDECS<TListingViewNode>*   decs  =
GetDeclarations(); decs->CalcSize(dc);
      TSortSTATS<TListingViewNode>*  stats =
GetStatements(); stats->CalcSize(dc);

      TSize    opensize, barsize, closesize;
      dc.GetTextExtent("|[", 2, opensize);
      dc.GetTextExtent("|",  1, barsize);
      dc.GetTextExtent("]|", 2, closesize);

      SetNodeSize(TSize(Indent+max(decs->NodeSize().cx,
stats->NodeSize().cx),
                        decs->NodeSize().cy+stats-
>NodeSize().cy+closesize.cy));
   }
}


void  TPROG<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int         y = pos.y;

      TSortDECS<TListingViewNode>*   decs  =
GetDeclarations();
      TSortSTATS<TListingViewNode>*  stats =
GetStatements();
      dc.TextOut(pos, "|[");
      decs->Draw(dc, TPoint(pos.x+Indent, y));
      y += decs->NodeHeight();
      dc.TextOut(pos.x, y, "|");
      stats->Draw(dc, TPoint(pos.x+Indent, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "]|");
   }
}


void  TDECS<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSize    semicolonsize;
      dc.GetTextExtent(";",     1, semicolonsize);

      int width       = 0;
      int height      = 0;
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TSortDEC<TListingViewNode> *dec   =
GetDeclaration(i);
         dec->CalcSize(dc);
         int sw = dec->NodeSize().cx+semicolonsize.cx;
         if(sw>width) width = sw;
         height += dec->NodeSize().cy;
      }
      SetNodeSize(TSize(width, height));
   }
}
void  TDECS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int y = pos.y;
      int numchildren = GetNumChildren();
      TSize    semicolonsize;
      dc.GetTextExtent(";",     1, semicolonsize);

      for(int i = 0; i<numchildren; i++) {
         TSortDEC<TListingViewNode> *dec   =
GetDeclaration(i);
         if(i) dc.TextOut(pos.x, y, ";");
         dec->Draw(dc, TPoint(pos.x+semicolonsize.cx,
y));
         y += dec->NodeSize().cy;
      }
   }
}

void  TListingViewNode::CalcSizeDelimiters(TMyDC& dc,
                                           char *start,
char *between, char *end)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSize    startsize(0,0), betweensize(0,0),
endsize(0,0);
      if(start)  dc.GetTextExtent(start,
   strlen(start), startsize);
      if(between)dc.GetTextExtent(between,
   strlen(between),  betweensize);
      if(end)    dc.GetTextExtent(end,     strlen(end),
      endsize);

      int width       = startsize.cx;
      int height      = max(max(startsize.cy,
betweensize.cy), endsize.cy);
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TListingViewNode *child   = GetChild(i);
         child->CalcSize(dc);
         width += child->NodeWidth() +(i<numchildren-
1?betweensize.cx:0);
         if(child->NodeHeight() > height)
            height = child->NodeHeight();
      }
      width += endsize.cx;
      SetNodeSize(TSize(width, height));
   }
}
void  TListingViewNode::DrawDelimiters(TMyDC& dc,
TPoint& pos,
                                           char *start,
char *between, char *end)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSize    startsize(0,0), betweensize(0,0),
endsize(0,0);
      if(start)  dc.GetTextExtent(start,
   strlen(start), startsize);
      if(between)dc.GetTextExtent(between,
   strlen(between),  betweensize);
      if(end)    dc.GetTextExtent(end,     strlen(end),
      endsize);

      int numchildren = GetNumChildren();

      if(start)dc.TextOut(pos, start);
      int x = pos.x + startsize.cx;
      for(int i = 0; i<numchildren; i++) {
         TListingViewNode *child   = GetChild(i);
         child->Draw(dc, TPoint(x, pos.y));
         x += child->NodeWidth();
         if(i<numchildren-1 && between) {
            dc.TextOut(x, pos.y, between); x +=
betweensize.cx;
         }
      }
      if(end)dc.TextOut(x, pos.y, end);
   }
}


void  TNDECS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "var  ", ", ", 0);
}
void  TNDECS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "var  ", ", ", 0);
}
void  TNDEC<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc,  0, ":", 0);
}
void  TNDEC<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ":", 0);
}

void  TPROCDEC<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortID<TListingViewNode>    *id = GetProcName();
      id->CalcSize(dc);
      TSortPBODY<TListingViewNode> *pbody = GetBody();
      pbody->CalcSize(dc);

      TSize    procsize, eqsize;
      dc.GetTextExtent("proc ", 5, procsize);
      dc.GetTextExtent(" = ",   3, eqsize);

      SetNodeSize(TSize(procsize.cx + id->NodeWidth() +
eqsize.cx + pbody->NodeWidth(),
                        pbody->NodeHeight()));
   }
}

void  TPROCDEC<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSize    procsize, eqsize;
      dc.GetTextExtent("proc ", 5, procsize);
      dc.GetTextExtent(" = ",   3, eqsize);
      dc.TextOut(pos, "proc");
      int   x = pos.x+procsize.cx;
      TSortID<TListingViewNode>    *id = GetProcName();
      id->Draw(dc, TPoint(x, pos.y));
      x += id->NodeWidth(); dc.TextOut(x, pos.y, " = ");
      x += eqsize.cx;
      TSortPBODY<TListingViewNode> *pbody = GetBody();
      pbody->Draw(dc, TPoint(x, pos.y));
   }
}


void  TFUNCDEC<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortID<TListingViewNode>    *id = GetFuncName();
      id->CalcSize(dc);
      TSortFBODY<TListingViewNode> *fbody = GetBody();
      fbody->CalcSize(dc);

      TSize    FUNCsize, eqsize;
      dc.GetTextExtent("func ", 5, FUNCsize);
      dc.GetTextExtent(" = ",   3, eqsize);

      SetNodeSize(TSize(FUNCsize.cx + id->NodeWidth() +
eqsize.cx + fbody->NodeWidth(),
                        fbody->NodeHeight()));
   }
}

void  TFUNCDEC<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSize    FUNCsize, eqsize;
      dc.GetTextExtent("func ", 5, FUNCsize);
      dc.GetTextExtent(" = ",   3, eqsize);
      dc.TextOut(pos, "func");
      int   x = pos.x+FUNCsize.cx;
      TSortID<TListingViewNode>    *id = GetFuncName();
      id->Draw(dc, TPoint(x, pos.y));
      x += id->NodeWidth(); dc.TextOut(x, pos.y, " = ");
      x += eqsize.cx;
      TSortFBODY<TListingViewNode> *fbody = GetBody();
      fbody->Draw(dc, TPoint(x, pos.y));
   }
}

void  TID<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TID<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, (char *)GetData()); }

void  TINTTYPE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "int"); }
void  TINTTYPE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "int"); }
void  TCHARTYPE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "char"); }
void  TCHARTYPE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "char"); }
void  TREALTYPE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "real"); }
void  TREALTYPE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "real"); }
void  TBOOLTYPE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "bool"); }
void  TBOOLTYPE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "bool"); }

void  TSTATS<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSize    semicolonsize;
      dc.GetTextExtent(";", 1, semicolonsize);

      int width       = 0;
      int height      = 0;
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TSortSTAT<TListingViewNode> *stat   =
GetStatement(i);
         stat->CalcSize(dc);
         int sw = semicolonsize.cx+stat->NodeSize().cx;
         if(sw>width)
            width = sw;
         height += stat->NodeSize().cy;
      }
      SetNodeSize(TSize(width, height));
   }
}

void  TSTATS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSize    semicolonsize;
      dc.GetTextExtent(";", 1, semicolonsize);
      int y = pos.y;
      int numchildren = GetNumChildren();
      for(int i = 0; i<numchildren; i++) {
         TSortSTAT<TListingViewNode>  *stat =
GetStatement(i);
         if(i)dc.TextOut(pos.x, y, ";");
         stat->Draw(dc, TPoint(pos.x+semicolonsize.cx,
y));
         y += stat->NodeSize().cy;
      }
   }
}

void  TSKIP<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "skip"); }
void  TSKIP<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, "skip"); }

void  TCONCASSIGN<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, " := ", 0);
}
void  TCONCASSIGN<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, 0, " := ", 0);
}


void  TPROCCALL<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      GetProcName()->CalcSize(dc);
      SetNodeSize(GetProcName()->NodeSize());
   }
}
void  TPROCCALL<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded())
      GetProcName()->Draw(dc, pos);
}

void  TPROCCALLARGS<TListingViewNode>::CalcSize(TMyDC&
dc)
{  CalcSizeDelimiters(dc, 0, "(", ")");
}
void  TPROCCALLARGS<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, 0, "(", ")");
}
void  TIFTHEN<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TListingViewNode>  *guard = GetGuard();
guard->CalcSize(dc);
      TSortSTATS<TListingViewNode> *stats =
GetTrueStatements(); stats->CalcSize(dc);
      TSize    ifsize, thensize, fisize;
      dc.GetTextExtent("if ", 3, ifsize);
      dc.GetTextExtent(" then", 5, thensize);
      dc.GetTextExtent("fi ", 3, fisize);
      SetNodeSize(TSize(max((int)(ifsize.cx+guard-
>NodeWidth()+thensize.cx), Indent+stats->NodeWidth()),
                        ifsize.cy+stats-
>NodeHeight()+fisize.cy));
   }
}

void  TIFTHEN<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSortEXPR<TListingViewNode>  *guard = GetGuard();
      TSortSTATS<TListingViewNode> *stats =
GetTrueStatements();
      TSize    ifsize, thensize, fisize;
      dc.GetTextExtent("if ", 3, ifsize);
      dc.GetTextExtent(" then", 5, thensize);
      dc.GetTextExtent("fi ", 3, fisize);
      int x = pos.x+ifsize.cx;
      dc.TextOut(pos, "if ");
      guard->Draw(dc, TPoint(x, pos.y));
      x += guard->NodeWidth();
      dc.TextOut(x, pos.y, " then");
      int y = pos.y+ifsize.cy;
      stats->Draw(dc, TPoint(pos.x+Indent, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "fi");
   }
}

void  TIFTHENELSE<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TListingViewNode>  *guard = GetGuard();
guard->CalcSize(dc);
      TSortSTATS<TListingViewNode> *tstats =
GetTrueStatements(); tstats->CalcSize(dc);
      TSortSTATS<TListingViewNode> *fstats =
GetFalseStatements(); fstats->CalcSize(dc);
      TSize    ifsize, thensize, elsesize, fisize;
      dc.GetTextExtent("if ", 3, ifsize);
      dc.GetTextExtent(" then", 5, thensize);
      dc.GetTextExtent("else", 4, elsesize);
      dc.GetTextExtent("fi", 2, fisize);
      SetNodeSize(TSize(max((int)(ifsize.cx+guard-
>NodeWidth()+thensize.cx),
                            Indent+max(tstats-
>NodeWidth(), fstats->NodeWidth())),
                        ifsize.cy+tstats-
>NodeHeight()+elsesize.cy+fstats-
>NodeHeight()+fisize.cy));
   }
}

void  TIFTHENELSE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSortEXPR<TListingViewNode>  *guard = GetGuard();
      TSortSTATS<TListingViewNode> *tstats =
GetTrueStatements();
      TSortSTATS<TListingViewNode> *fstats =
GetFalseStatements();
      TSize    ifsize, thensize, elsesize, fisize;
      dc.GetTextExtent("if ", 3, ifsize);
      dc.GetTextExtent(" then", 5, thensize);
      dc.GetTextExtent("else", 4, elsesize);
      dc.GetTextExtent("fi", 2, fisize);
      dc.TextOut(pos, "if ");
      int x = pos.x+ifsize.cx;
      guard->Draw(dc, TPoint(x, pos.y));
      x += guard->NodeWidth();
      dc.TextOut(x, pos.y, " then");
      int y = pos.y+ifsize.cy;
      tstats->Draw(dc, TPoint(pos.x+Indent, y));
      y += tstats->NodeHeight();
      dc.TextOut(pos.x, y, "else");
      y += elsesize.cy;
      fstats->Draw(dc, TPoint(pos.x+Indent, y));
      y += fstats->NodeHeight();
      dc.TextOut(pos.x, y, "fi");
   }
}


void  TWHILE<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TListingViewNode>  *guard = GetGuard();
guard->CalcSize(dc);
      TSortSTATS<TListingViewNode> *stats =
GetStatements(); stats->CalcSize(dc);
      TSize    whilesize, dosize, odsize;
      dc.GetTextExtent("while ", 6, whilesize);
      dc.GetTextExtent(" do", 3, dosize);
      dc.GetTextExtent("od", 2, odsize);
      SetNodeSize(TSize(max((int)(whilesize.cx+guard-
>NodeWidth()+dosize.cx), Indent+stats->NodeWidth()),
                        whilesize.cy+stats-
>NodeHeight()+odsize.cy));
   }
}

void  TWHILE<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSortEXPR<TListingViewNode>  *guard = GetGuard();
      TSortSTATS<TListingViewNode> *stats =
GetStatements();
      TSize    whilesize, dosize, odsize;
      dc.GetTextExtent("while ", 6, whilesize);
      dc.GetTextExtent(" do", 3, dosize);
      dc.GetTextExtent("od", 2, odsize);
      int x = pos.x+whilesize.cx;
      dc.TextOut(pos, "while ");
      guard->Draw(dc, TPoint(x, pos.y));
      x += guard->NodeWidth();
      dc.TextOut(x, pos.y, " do");
      int y = pos.y+whilesize.cy;
      stats->Draw(dc, TPoint(pos.x+Indent, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "od");
   }
}



void  TREAD<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "read(", ",", ")");
}
void  TREAD<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "read(", ",", ")");
}

void  TWRITE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "write(", ",", ")");
}
void  TWRITE<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "write(", ",", ")");
}

void  TFOR<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortID<TListingViewNode>    *id  = GetVariable();
         id->CalcSize(dc);
      TSortEXPR<TListingViewNode>  *from   =
GetFromExpression(); from->CalcSize(dc);
      TSortEXPR<TListingViewNode>  *by     =
GetByExpression();   by->CalcSize(dc);
      TSortEXPR<TListingViewNode>  *to     =
GetToExpression();   to->CalcSize(dc);
      TSortSTATS<TListingViewNode> *stats =
GetStatements();     stats->CalcSize(dc);
      TSize    forsize, fromsize, bysize, tosize,
dosize, odsize;
      dc.GetTextExtent("for ",   4, forsize);
      dc.GetTextExtent(" from ", 6, fromsize);
      dc.GetTextExtent(" by ",   4, bysize);
      dc.GetTextExtent(" to ",   4, tosize);
      dc.GetTextExtent(" do",  3, dosize);
      dc.GetTextExtent("od",      2, odsize);
      SetNodeSize(TSize(max((int)(forsize.cx+id-
>NodeWidth()+
                            fromsize.cx+from-
>NodeWidth()+
                            bysize.cx + by-
>NodeWidth()+
                            tosize.cx + to-
>NodeWidth()+dosize.cx),
                            Indent+stats->NodeWidth()),
                        forsize.cy+stats-
>NodeHeight()+odsize.cy));
   }
}

void  TFOR<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSortID<TListingViewNode>    *id  = GetVariable();
      TSortEXPR<TListingViewNode>  *from   =
GetFromExpression();
      TSortEXPR<TListingViewNode>  *by     =
GetByExpression();
      TSortEXPR<TListingViewNode>  *to     =
GetToExpression();
      TSortSTATS<TListingViewNode> *stats =
GetStatements();
      TSize    forsize, fromsize, bysize, tosize,
dosize, odsize;
      dc.GetTextExtent("for ",   4, forsize);
      dc.GetTextExtent(" from ", 6, fromsize);
      dc.GetTextExtent(" by ",   4, bysize);
      dc.GetTextExtent(" to ",   4, tosize);
      dc.GetTextExtent(" do", 3, dosize);
      dc.GetTextExtent("od",      2, odsize);

      dc.TextOut(pos, "for ");
      int x = pos.x+forsize.cx;
      id->Draw(dc, TPoint(x, pos.y));
      x += id->NodeWidth();
      dc.TextOut(x, pos.y, " from ");
      x += fromsize.cx;
      from->Draw(dc, TPoint(x, pos.y));
      x += from->NodeWidth();
      dc.TextOut(x, pos.y, " by ");
      x += bysize.cx;
      by->Draw(dc, TPoint(x, pos.y));
      x += by->NodeWidth();
      dc.TextOut(x, pos.y, " to ");
      x += tosize.cx;
      to->Draw(dc, TPoint(x, pos.y));
      x += to->NodeWidth();
      dc.TextOut(x, pos.y, " do");

      int y = pos.y+forsize.cy;
      stats->Draw(dc, TPoint(pos.x+Indent, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "od");
   }
}

void  TBLOCK<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortDECS<TListingViewNode>*   decs  =
GetDeclarations(); decs->CalcSize(dc);
      TSortSTATS<TListingViewNode>*  stats =
GetStatements(); stats->CalcSize(dc);

      TSize    opensize, barsize, closesize;
      dc.GetTextExtent("|[", 2, opensize);
      dc.GetTextExtent("|",  1, barsize);
      dc.GetTextExtent("]|", 2, closesize);

      SetNodeSize(TSize(Indent+max(decs->NodeSize().cx,
stats->NodeSize().cx),
                        decs->NodeSize().cy+stats-
>NodeSize().cy+closesize.cy));
   }
}


void  TBLOCK<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int         y = pos.y;

      TSortDECS<TListingViewNode>*   decs  =
GetDeclarations();
      TSortSTATS<TListingViewNode>*  stats =
GetStatements();
      dc.TextOut(pos, "|[");
      decs->Draw(dc, TPoint(pos.x+Indent, y));
      y += decs->NodeHeight();
      dc.TextOut(pos.x, y, "|");
      stats->Draw(dc, TPoint(pos.x+Indent, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "]|");
   }
}


void  TVARS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TVARS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TEXPRS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc,  0, ", ", 0);
}
void  TEXPRS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TVARVALUE<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      GetVariable()->CalcSize(dc);
SetNodeSize(GetVariable()->NodeSize());
   }
}

void  TVARVALUE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded())
      GetVariable()->Draw(dc, pos);
}
void  TFUNCCALL<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, 0, 0);
}
void  TFUNCCALL<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, 0, 0, 0);
}

void  TFUNCCALLARGS<TListingViewNode>::CalcSize(TMyDC&
dc)
{  CalcSizeDelimiters(dc, 0, "(", ")");
}
void  TFUNCCALLARGS<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, 0, "(", ")");
}

void  TNUMVALUE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TNUMVALUE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, (char *)GetData()); }

void  TCHARVALUE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TCHARVALUE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, (char *)GetData()); }

void  TIMPLICATION<TListingViewNode>::CalcSize(TMyDC&
dc)
{  CalcSizeDelimiters(dc, "(", " => ", ")");
}
void  TIMPLICATION<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " => ", ")");
}
void  TEQUIVALENCE<TListingViewNode>::CalcSize(TMyDC&
dc)
{  CalcSizeDelimiters(dc, "(", " == ", ")");
}
void  TEQUIVALENCE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " == ", ")");
}

void  TOR<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " \\/ ", ")");
}
void  TOR<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " \\/ ", ")");
}
```

```
void  TAND<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " /\\ ", ")");
}
void  TAND<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " /\\ ", ")");
}

void  TLESS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " < ", ")");
}
void  TLESS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " < ", ")");
}


void  TLESSEQ<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " <= ", ")");
}
void  TLESSEQ<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " <= ", ")");
}


void  TGREATER<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " > ", ")");
}
void  TGREATER<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " > ", ")");
}


void  TGREATEREQ<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " >= ", ")");
}
void  TGREATEREQ<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " >= ", ")");
}


void  TEQUALITY<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " == ", ")");
}
void  TEQUALITY<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " == ", ")");
}

void  TDIFFERENCE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " /= ", ")");
}
void  TDIFFERENCE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " /= ", ")");
}

void  TADDITION<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " + ", ")");
}
void  TADDITION<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " + ", ")");
}

void  TSUBTRACTION<TListingViewNode>::CalcSize(TMyDC&
dc)
{  CalcSizeDelimiters(dc, "(", " - ", ")");
}
void  TSUBTRACTION<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " - ", ")");
}

void  TMULTIPLICATION<TListingViewNode>::CalcSize(TMyDC&
dc)
{  CalcSizeDelimiters(dc, "(", "*", ")");
}
void  TMULTIPLICATION<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", "*", ")");
}

void  TDIVISION<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", "/", ")");
}
void  TDIVISION<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", "/", ")");
}

void  TMODULO<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " mod ", ")");
}
void  TMODULO<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " mod ", ")");
}


void  TUNARYMINUS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "-", 0, 0);
}
void  TUNARYMINUS<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "-", 0, 0);
}

void  TNOT<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "NOT ", 0, 0);
}
void  TNOT<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "NOT ", 0, 0);
}

void  TEXPRBLOCK<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortDECS<TListingViewNode>*   decs  =
GetDeclarations(); decs->CalcSize(dc);
      TSortSTATS<TListingViewNode>*  stats =
GetStatements();   stats->CalcSize(dc);
      TSortEXPR<TListingViewNode>*   expr  =
GetExpression();   expr->CalcSize(dc);

      TSize    opensize, barsize, closesize;
      dc.GetTextExtent("|[", 2, opensize);
      dc.GetTextExtent("|",  1, barsize);
      dc.GetTextExtent("]|", 2, closesize);

      SetNodeSize(TSize(Indent+max((int)(max(decs-
>NodeSize().cx, stats->NodeSize().cx)), expr-
>NodeWidth()),
                        decs->NodeSize().cy+stats-
>NodeSize().cy+expr->NodeHeight()+closesize.cy));
   }
}
void  TEXPRBLOCK<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int         y = pos.y;

      TSortDECS<TListingViewNode>*   decs  =
GetDeclarations();
      TSortSTATS<TListingViewNode>*  stats =
GetStatements();
      TSortEXPR<TListingViewNode>*   expr  =
GetExpression();

      dc.TextOut(pos, "|[");
      decs->Draw(dc, TPoint(pos.x+Indent, y));
      y += decs->NodeHeight();
      dc.TextOut(pos.x, y, "|");
      stats->Draw(dc, TPoint(pos.x+Indent, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "|");
      expr->Draw(dc, TPoint(pos.x+Indent, y));
      y += expr->NodeHeight();
      dc.TextOut(pos.x, y, "]|");
   }
}

void  TPBODYSTATS<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSize    semicolonsize;
      dc.GetTextExtent(";", 1, semicolonsize);

      int width       = 0;
      int height      = semicolonsize.cy; // for closing
bracket
      int numchildren = GetNumChildren();
      for(int i = 0; i<numchildren; i++) {
         TSortSTAT<TListingViewNode> *stat   =
GetStatement(i);
         stat->CalcSize(dc);
         int sw = Indent+semicolonsize.cx+stat-
>NodeSize().cx;
         if(sw>width)
            width = sw;
         height += stat->NodeSize().cy;
      }
      SetNodeSize(TSize(width, height));
   }
}

void  TPBODYSTATS<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      TSize    semicolonsize;
      dc.GetTextExtent(";", 1, semicolonsize);
      TEXTMETRIC tm;
      dc.GetTextMetrics(tm);
      int y = pos.y;
      int numchildren = GetNumChildren();
      dc.TextOut(pos, "(");
      for(int i = 0; i<numchildren; i++) {
         TSortSTAT<TListingViewNode>  *stat =
GetStatement(i);
         if(i)
            dc.TextOut(pos.x+Indent, y, ";");
         stat->Draw(dc,
TPoint(pos.x+Indent+semicolonsize.cx, y));
         y += stat->NodeSize().cy;
      }
      dc.TextOut(pos.x, y, ")");
   }
}


void  TPBODYPARSSTATS<TListingViewNode>::CalcSize(TMyDC&
dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortPPARS<TListingViewNode>*  pars  =
GetProcParameters(); pars->CalcSize(dc);
      TSortSTATS<TListingViewNode>*  stats =
GetStatements(); stats->CalcSize(dc);

      TSize    opensize, barsize, closesize;
      dc.GetTextExtent("(", 1, opensize);
      dc.GetTextExtent("|", 1, barsize);
      dc.GetTextExtent(")", 1, closesize);

      SetNodeSize(TSize(Indent+max(pars->NodeSize().cx,
stats->NodeSize().cx),
                        pars->NodeSize().cy+stats-
>NodeSize().cy+closesize.cy));
   }
}
void  TPBODYPARSSTATS<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int         y = pos.y;

      TSortPPARS<TListingViewNode>*  pars  =
GetProcParameters();
      TSortSTATS<TListingViewNode>*  stats =
GetStatements();
      dc.TextOut(pos, "(");
      pars->Draw(dc, TPoint(pos.x+Indent, y));
      y += pars->NodeHeight();
      dc.TextOut(pos.x, y, "|");
      stats->Draw(dc, TPoint(pos.x+Indent, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, ")");
   }
}

void  TPPARS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TPPARS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TPPARVALUE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ": ", 0);
}
void  TPPARVALUE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "? ", ": ", 0);
}

void  TPPARREF<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "! ", ": ", 0);
}
void  TPPARREF<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "! ", ": ", 0);
}

void  TPARGS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TPARGS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TPARGVALUE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ": ", 0);
}
void  TPARGVALUE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "? ", ": ", 0);
}

void  TPARGREF<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "! ", ": ", 0);
}
void  TPARGREF<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "! ", ": ", 0);
}

void  TFBODY<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortFPARS<TListingViewNode>*  pars  =
GetFuncParameters(); pars->CalcSize(dc);
      TSortEXPR<TListingViewNode>*   expr  =
GetExpression();     expr->CalcSize(dc);

      TSize    opensize, barsize, closesize;
      dc.GetTextExtent("(", 1, opensize);
      dc.GetTextExtent("|", 1, barsize);
      dc.GetTextExtent(")", 1, closesize);

      SetNodeSize(TSize(Indent+max(pars->NodeSize().cx,
expr->NodeSize().cx),
                        pars->NodeSize().cy+expr-
>NodeSize().cy+closesize.cy));
   }
}
void  TFBODY<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int         y = pos.y;

      TSortFPARS<TListingViewNode>*  pars  =
GetFuncParameters();
      TSortEXPR<TListingViewNode>*   expr  =
GetExpression();

      dc.TextOut(pos, "(");
      pars->Draw(dc, TPoint(pos.x+Indent, y));
      y += pars->NodeHeight();
      dc.TextOut(pos.x, y, "|");
      expr->Draw(dc, TPoint(pos.x+Indent, y));
      y += expr->NodeHeight();
      dc.TextOut(pos.x, y, ")");
   }
}

void  TFPARS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TFPARS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TFPARVALUE<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ": ", 0);
}
```

```
void  TFPARVALUE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "? ", ": ", 0);
}

void  TFARGS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ", ? ", 0);
}
void  TFARGS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "? ", ", ? ", 0);
}

void  TNUMCASE<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TListingViewNode>*   expr  = GetCase();
expr->CalcSize(dc);
      TSortALTS1<TListingViewNode>*  alts  = GetIn();
alts->CalcSize(dc);
      TSortSTATS<TListingViewNode>*  stats = GetOut();
stats->CalcSize(dc);

      TSize    numcasesize, insize, esacsize;
      dc.GetTextExtent("numcase ",   8, numcasesize);
      dc.GetTextExtent(" in",        3, insize);
      dc.GetTextExtent("esac",       4, esacsize);

      SetNodeSize(TSize(max(numcasesize.cx+expr-
>NodeWidth()+insize.cx,
                            Indent+max(alts-
>NodeWidth(), Indent+stats->NodeWidth())),
                        expr->NodeHeight()+alts-
>NodeHeight()+stats->NodeHeight()+
                        esacsize.cy));
   }
}

void  TNUMCASE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int         y = pos.y;
      TSortEXPR<TListingViewNode>*   expr  = GetCase();
      TSortALTS1<TListingViewNode>*  alts  = GetIn();
      TSortSTATS<TListingViewNode>*  stats = GetOut();
      TSize    numcasesize;
      dc.GetTextExtent("numcase ",   8, numcasesize);

      dc.TextOut(pos, "numcase");
      expr->Draw(dc, TPoint(pos.x+numcasesize.cx, y));
      dc.TextOut(pos.x+numcasesize.cx+expr->NodeWidth(),
y, " in");
      y += expr->NodeHeight();
      alts->Draw(dc, TPoint(pos.x+Indent, y));
      y += alts->NodeHeight();
      dc.TextOut(pos.x+Indent, y, "out");
      stats->Draw(dc, TPoint(pos.x+Indent*2, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "esac");
   }
}

void  TLABCASE<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TListingViewNode>*   expr  = GetCase();
expr->CalcSize(dc);
      TSortALTS2<TListingViewNode>*  alts  = GetIn();
alts->CalcSize(dc);
      TSortSTATS<TListingViewNode>*  stats = GetOut();
stats->CalcSize(dc);

      TSize    labcasesize, insize, esacsize;
      dc.GetTextExtent("labcase ",   8, labcasesize);
      dc.GetTextExtent(" in",        3, insize);
      dc.GetTextExtent("esac",       4, esacsize);

      SetNodeSize(TSize(max(labcasesize.cx+expr-
>NodeWidth()+insize.cx,
                            Indent+max(alts-
>NodeWidth(), Indent+stats->NodeWidth())),
                        expr->NodeHeight()+alts-
>NodeHeight()+stats->NodeHeight()+
                        esacsize.cy));
   }
}

void  TLABCASE<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int         y = pos.y;
      TSortEXPR<TListingViewNode>*   expr  = GetCase();
      TSortALTS2<TListingViewNode>*  alts  = GetIn();
      TSortSTATS<TListingViewNode>*  stats = GetOut();
      TSize    labcasesize;
      dc.GetTextExtent("labcase ",   8, labcasesize);

      dc.TextOut(pos, "labcase");
      expr->Draw(dc, TPoint(pos.x+labcasesize.cx, y));
      dc.TextOut(pos.x+labcasesize.cx+expr->NodeWidth(),
y, " in");
      y += expr->NodeHeight();
      alts->Draw(dc, TPoint(pos.x+Indent, y));
      y += alts->NodeHeight();
      dc.TextOut(pos.x+Indent, y, "out");
      stats->Draw(dc, TPoint(pos.x+Indent*2, y));
      y += stats->NodeHeight();
      dc.TextOut(pos.x, y, "esac");
   }
}

void  TALTS1<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      int width       = Indent;
      int height      = 0;
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TSortSTATS<TListingViewNode> *stats   =
GetStatements(i);
         stats->CalcSize(dc);
         int sw = Indent+stats->NodeSize().cx;
         if(sw>width)
            width = sw;
         height += stats->NodeSize().cy;
      }
      SetNodeSize(TSize(width, height));
   }
}

void  TALTS1<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int y = pos.y;
      int numchildren = GetNumChildren();
      for(int i = 0; i<numchildren; i++) {
         TSortSTATS<TListingViewNode>  *stats =
GetStatements(i);
         if(i)dc.TextOut(pos.x, y, "[]");
         stats->Draw(dc, TPoint(Indent+pos.x, y));
         y += stats->NodeSize().cy;
      }
   }
}

void  TALTS2<TListingViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      int width       = Indent;
      int height      = 0;
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TSortALT2<TListingViewNode> *alt2   =
GetAlternative(i);
         alt2->CalcSize(dc);
         int sw = Indent+alt2->NodeSize().cx;
         if(sw>width)
            width = sw;
         height += alt2->NodeSize().cy;
      }
      SetNodeSize(TSize(width, height));
   }
}

void  TALTS2<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(!GetFolded()) {
      int y = pos.y;
      int numchildren = GetNumChildren();
      for(int i = 0; i<numchildren; i++) {
         TSortALT2<TListingViewNode>  *alt2 =
GetAlternative(i);
         if(i)dc.TextOut(pos.x, y, "[]");
         alt2->Draw(dc, TPoint(Indent+pos.x, y));
         y += alt2->NodeSize().cy;
      }
   }
}
void  TALT2<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ": ", 0);
}
void  TALT2<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ": ", 0);
}
void  TLABELS<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ",", 0);
}
void  TLABELS<TListingViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ",", 0);
}
void  TNUMLABEL<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TNUMLABEL<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, (char *)GetData()); }

void  TCHARLABEL<TListingViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TCHARLABEL<TListingViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, (char *)GetData()); }

typedef TScrolledTreeWindowView<TListingViewNode>
TListingView;

DEFINE_RESPONSE_TABLE1(TListingView,
TScrolledWindowView)
   EV_WM_CHAR,
   EV_WM_LBUTTONDOWN,
   EV_WM_RBUTTONDOWN,
   EV_WM_LBUTTONDBLCLK,
   EV_WM_KEYDOWN,
   EV_WM_KEYUP,
   EV_WM_SETFOCUS,
   EV_VN_MIRRORSUBTREE,
   EV_VN_GLOBALFOCUS,

   EV_COMMAND(CM_PREVNODE,       PrevNode),
   EV_COMMAND(CM_NEXTNODE,       NextNode),
   EV_COMMAND(CM_PREVHOLE,       PrevHole),
   EV_COMMAND(CM_NEXTHOLE,       NextHole),
   EV_COMMAND(CM_DELETE,         Delete),
   EV_COMMAND(CM_CUT,            Cut),
   EV_COMMAND(CM_COPY,           Copy),
   EV_COMMAND(CM_PASTE,          Paste),
   EV_COMMAND(CM_PARENT,         Parent),
   EV_COMMAND(CM_FOLD,           ToggleFold),
   EV_COMMAND(CM_ADDHOLETOLIST,  AddHoleToList),
   EV_COMMAND_AND_ID(CM_ADDHOLEBEFORE,
AddHoleBeforeAfter),
   EV_COMMAND_AND_ID(CM_ADDHOLEAFTER,
AddHoleBeforeAfter),
END_RESPONSE_TABLE;
const char far*
TScrolledTreeWindowView<TListingViewNode>::StaticName()
{
   return "Listing View";
}

void TListingView::Paint(TDC& dc, BOOL, TRect&)
{
   dc.SetBkMode(TRANSPARENT);
   TListingViewNode::ListingTree = &Tree;
   if(Tree.GetRoot()) {
      Tree.GetRoot()->FlushNodePositions();
      Tree.GetRoot()->Draw((TMyDC&)dc, TPoint(0, 0));
   }
}


DEFINE_DOC_TEMPLATE_CLASS(TStructDocument, TListingView,
TListingTemplate);
TListingTemplate listingTpl("Listing View", "*.STR", 0,
"STR",
                            dtAutoDelete | dtUpdateDir
|dtAutoOpen);
```

flow_pl4.cpp

```
//---------------------------------------------------------
---------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TFlowView
//---------------------------------------------------------
---------------------
#include "owlhdr.h"
#pragma hdrstop

//#pragma option -Jgx

#include <owl\menu.h>

#include "views.rc"
#include "tree.h"
#include "strucdoc.h"
#include "mydc.h"
#include "viewnode.h"
#include "viewtree.h"
#include "scrwview.h"
#include "scrtrwvw.h"


class   TFlowViewNode : public TViewNode {
public:
                         TFlowViewNode(int n = 0, int
fixed = FALSE);
   inline TFlowViewNode *GetChild(int n) const;
   void                 CalcSizeFolded(TMyDC&);
   void                 DrawFolded(TMyDC&, TPoint&
pos);
   void                  CalcSizeString(TMyDC&, const
char const *);
   void                  DrawString(TMyDC&, TPoint&,
const char const *);
   void                  CalcSizeHole(TMyDC&);
   void                  DrawHole(TMyDC&, TPoint&);
   BOOL                  DoPosFocusFolded(TMyDC& dc,
TPoint& pos);
   void               CalcSizeDelimiters(TMyDC& dc, \
                                          char *start,
char *between, char *end);

   void               DrawDelimiters(TMyDC& dc, TPoint&
pos, \
                                          char *start,
char *between, char *end);
   void                  DrawStatBox(TMyDC& dc, TPoint&
pos);
   inline int           GetConnX() const;
   inline void          SetConnX(int connx);

   static TViewTree<TFlowViewNode>     *FlowTree;

private:
   int                  ConnX;
};

inline TFlowViewNode* TFlowViewNode::GetChild(int n)
const
{  return (TFlowViewNode*)TTreeNode::GetChild(n);
}
inline int             TFlowViewNode::GetConnX() const
{  return ConnX; }
inline void            TFlowViewNode::SetConnX(int
connx)
{  ConnX = connx; }



const int        DistX       = 6;
const int        DistY       = 8;
const int        TextBorderX = 6;
const int        TextBorderY = 2;
const int        BlockBorderX = 4;
const int        BlockBorderY = 4;
const int        HConnLen    = 6; //length of horiz
connecting line
const int        VConnLen    = 6; //length of horiz
connecting line
const int      BoxExtraX   = TextBorderX*2+RShadW;
const int      BoxExtraY= TextBorderY*2+BShadH;

#define    LineColor    TColor(0,0,0)
#define    FocusColor   TColor(128, 128, 128)
#define    FillColor    TColor(255, 255, 255)
#define    ScopeColor   TColor(230, 230, 240)
```

```
static TPen        ScopePen(ScopeColor, 1, PS_DOT);

static TPen        Pen(LineColor);
static TBrush      FillBrush(FillColor);
static TBrush      FocusBrush(FocusColor);
static TBrush      ShadowBrush(TColor::Black);

TViewTree<TFlowViewNode> *TFlowViewNode::FlowTree = 0;

TFlowViewNode::TFlowViewNode(int n, int fixed):
   ConnX(0),TViewNode(n, fixed)
{
}

void               TFlowViewNode::CalcSizeString(TMyDC&
dc, const char const *s)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      SetNodeSize(dc.GetTextExtent(s, strlen(s)));
      SetConnX(NodeWidth()/2);
   }
}

void
TFlowViewNode::DrawString(TMyDC& dc, TPoint& pos, const
char const *s)
{
   if(DoPosFocusFolded(dc, pos)) {
      dc.TextOut(pos, s);
   }
}
BOOL
   TFlowViewNode::DoPosFocusFolded(TMyDC& dc, TPoint&
pos)
{
   TRect    bounding(pos, NodeSize());

   AddNodePosition(pos);
   if(!bounding.Touches(dc.GetClipBox()))
      return FALSE;
   FlushChildrenNodePositions();
   if(FlowTree->GetFocus() == this)
      dc.FillRect(bounding, FocusBrush);
   if(GetFolded()) {
      DrawFolded(dc, pos);
      return FALSE;
   }
   return TRUE;
}
void              TFlowViewNode::CalcSizeFolded(TMyDC&
dc)
{
   char  str[80];

   wsprintf(str, "%s <%s>", GetOpName(), GetSortName());
   TSize    sz = dc.GetTextExtent(str,
strlen(str))+TSize(PlusSize+DistX, 0);
   if(GetSort() == SORT_STAT)
      sz += TSize(BoxExtraX, BoxExtraY);
   SetNodeSize(sz);
   SetConnX(sz.cx/2);
}


void              TFlowViewNode::DrawFolded(TMyDC& dc,
TPoint& p2)
{
   char  str[80];
   TPoint pos = p2;

   if(GetSort() == SORT_STAT) {
      DrawStatBox(dc, p2);
      pos = p2+TSize(TextBorderX, TextBorderY);
   }
   wsprintf(str, "%s <%s>", GetOpName(), GetSortName());
   dc.Plus(TPoint(pos.x, p2.y+(NodeSize().cy-
PlusSize)/2));
   dc.TextOut(pos.x+PlusSize+DistX, pos.y, str);
}


void  TFlowViewNode::CalcSizeDelimiters(TMyDC& dc,
                                          char *start,
char *between, char *end)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSize    startsize(0,0), betweensize(0,0),
endsize(0,0);
      if(start)  dc.GetTextExtent(start,
   strlen(start), startsize);
      if(between)dc.GetTextExtent(between,
   strlen(between),  betweensize);
      if(end)    dc.GetTextExtent(end,     strlen(end),
      endsize);

      int width       = startsize.cx;
      int height      = max(max(startsize.cy,
betweensize.cy), endsize.cy);
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TFlowViewNode *child   = GetChild(i);
         child->CalcSize(dc);
         width += child->NodeWidth() +(i<numchildren-
1?betweensize.cx:0);
         if(child->NodeHeight() > height)
            height = child->NodeHeight();
      }
      width += endsize.cx;
      if(GetSort() == SORT_STAT) {
         width += BoxExtraX; height += BoxExtraY;
      }
      SetNodeSize(TSize(width, height));
      SetConnX(width/2);
   }
}
void  TFlowViewNode::DrawDelimiters(TMyDC& dc, TPoint&
pos,
                                          char *start,
char *between, char *end)
{  if(DoPosFocusFolded(dc, pos)) {
      TSize    startsize(0,0), betweensize(0,0),
endsize(0,0);
      if(start)  dc.GetTextExtent(start,
   strlen(start), startsize);
      if(between)dc.GetTextExtent(between,
   strlen(between),  betweensize);
      if(end)    dc.GetTextExtent(end,     strlen(end),
      endsize);

      int numchildren = GetNumChildren();

      int x = pos.x;
      int y = pos.y;

      if(GetSort() == SORT_STAT) {
         DrawStatBox(dc, pos);
         x += TextBorderX; y += TextBorderY;
      }

      if(start)dc.TextOut(x, y, start);
      x += startsize.cx;
      for(int i = 0; i<numchildren; i++) {
         TFlowViewNode *child   = GetChild(i);
         child->Draw(dc, TPoint(x, y));
         x += child->NodeWidth();
         if(i<numchildren-1 && between) {
            dc.TextOut(x, y, between); x +=
betweensize.cx;
         }
      }
      if(end)dc.TextOut(x, y, end);
   }
}


void  TFlowViewNode::DrawStatBox(TMyDC& dc, TPoint& pos)
{
   TRect     rect(pos, NodeSize()-TSize(RShadW,
BShadH));
   dc.ShadowRectangle(rect, FlowTree->GetFocus() == this
? FocusBrush : FillBrush,
                       Pen, ShadowBrush);
}



void  TSTATHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   SetNodeSize(dc.GetTextExtent("? <STAT>",
8)+TSize(BoxExtraX, BoxExtraY));
   SetConnX(NodeWidth()/2);
}

void  TSTATHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  if(DoPosFocusFolded(dc, pos)) {
      DrawStatBox(dc, pos);
      dc.TextOut(pos.x+TextBorderX, pos.y+TextBorderY,
"? <STAT>");
   }
}


void                TFlowViewNode::CalcSizeHole(TMyDC&
dc)
{
   static char    s[40] = "? <";
   char  *p = GetSortName();
   strcpy(s+3, p); strcpy(s+3+strlen(p), ">");
   CalcSizeString(dc, s);
}

void                TFlowViewNode::DrawHole(TMyDC& dc,
TPoint& pos)
{
   static char    s[40] = "? <";
   char  *p = GetSortName();
   strcpy(s+3, p); strcpy(s+3+strlen(p), ">");
   DrawString(dc, pos, s);
}


void  TPROGHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPROGHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TDECHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TDECHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TDECSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TDECSHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TNDECHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TNDECHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TIDHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TIDHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TTYPEHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TTYPEHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
/*void  TSTATHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TSTATHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }*/
void  TSTATSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TSTATSHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TVARSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TVARSHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TEXPRSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TEXPRSHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TEXPRHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TEXPRHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TALTS1HOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALTS1HOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TALTS2HOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALTS2HOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
```

```
void  TALT2HOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALT2HOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TLABELSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TLABELSHOLE<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TLABELHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TLABELHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TPBODYHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPBODYHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TPPARSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPPARSHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TPPARHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPPARHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TPARGSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPARGSHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TPARGHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPARGHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TFBODYHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFBODYHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TFPARSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFPARSHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TFPARHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFPARHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TFARGSHOLE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFARGSHOLE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }


void  TPROG<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortDECS<TFlowViewNode>*   decs  =
GetDeclarations(); decs->CalcSize(dc);
      TSortSTATS<TFlowViewNode>*  stats =
GetStatements();  stats->CalcSize(dc);

      SetNodeSize(TSize(2*BlockBorderX+max(decs-
>NodeSize().cx, stats->NodeSize().cx),
                        2*BlockBorderY+decs-
>NodeSize().cy+ArrowLength+stats->NodeSize().cy));
      SetConnX(NodeWidth()/2);
   }
}


void  TPROG<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  if(DoPosFocusFolded(dc, pos)) {
      int         y = pos.y+BlockBorderY;

      TSortDECS<TFlowViewNode>*   decs  =
GetDeclarations();
      TSortSTATS<TFlowViewNode>*  stats =
GetStatements();

      dc.SelectObject(ScopePen);
      dc.RectangleOutline(TRect(pos, NodeSize()));
      dc.RestorePen();

      decs->Draw(dc, TPoint(pos.x+BlockBorderX, y));
      y += decs->NodeHeight()+ArrowLength;

      dc.Arrow(TPoint(pos.x+BlockBorderX+stats-
>GetConnX(), y), TMyDC::DOWN);
      stats->Draw(dc, TPoint(pos.x+BlockBorderX, y));
   }
}


void  TDECS<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSize    semicolonsize;
      dc.GetTextExtent(";",     1, semicolonsize);

      int width       = 0;
      int height      = 0;
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TSortDEC<TFlowViewNode> *dec   =
GetDeclaration(i);
         dec->CalcSize(dc);
         int sw = dec->NodeSize().cx+semicolonsize.cx;
         if(sw>width) width = sw;
         height += dec->NodeSize().cy;
      }
      SetNodeSize(TSize(width, height));
   }
}
void  TDECS<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  if(DoPosFocusFolded(dc, pos)) {
      int y = pos.y;
      int numchildren = GetNumChildren();
      TSize    semicolonsize;
      dc.GetTextExtent(";",     1, semicolonsize);

      for(int i = 0; i<numchildren; i++) {
         TSortDEC<TFlowViewNode> *dec   =
GetDeclaration(i);
         if(i) dc.TextOut(pos.x, y, ";");
         dec->Draw(dc, TPoint(pos.x+semicolonsize.cx,
y));
         y += dec->NodeSize().cy;
      }
   }
}


void  TNDECS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "var  ", ", ", 0);
}
void  TNDECS<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "var  ", ", ", 0);
}
void  TNDEC<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc,  0, ":", 0);
}
void  TNDEC<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, 0, ":", 0);
}

void  TPROCDEC<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortID<TFlowViewNode>    *id = GetProcName();
      id->CalcSize(dc);
      TSortPBODY<TFlowViewNode> *pbody = GetBody();
      pbody->CalcSize(dc);

      TSize    procsize, eqsize;
      dc.GetTextExtent("proc ", 5, procsize);
      dc.GetTextExtent(" = ",   3, eqsize);

      SetNodeSize(TSize(procsize.cx + id->NodeWidth() +
eqsize.cx + pbody->NodeWidth(),
                        pbody->NodeHeight()));
   }
}

void  TPROCDEC<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
```

```
{  if(DoPosFocusFolded(dc, pos)) {
      TSize    procsize, eqsize;
      dc.GetTextExtent("proc ", 5, procsize);
      dc.GetTextExtent(" = ",   3, eqsize);
      dc.TextOut(pos, "proc");
      int   x = pos.x+procsize.cx;
      TSortID<TFlowViewNode>   *id = GetProcName();
      id->Draw(dc, TPoint(x, pos.y));
      x += id->NodeWidth(); dc.TextOut(x, pos.y, " = ");
      x += eqsize.cx;
      TSortPBODY<TFlowViewNode> *pbody = GetBody();
      pbody->Draw(dc, TPoint(x, pos.y));
   }
}


void  TFUNCDEC<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortID<TFlowViewNode>   *id = GetFuncName();
      id->CalcSize(dc);
      TSortFBODY<TFlowViewNode> *fbody = GetBody();
      fbody->CalcSize(dc);

      TSize    FUNCsize, eqsize;
      dc.GetTextExtent("func ", 5, FUNCsize);
      dc.GetTextExtent(" = ",   3, eqsize);

      SetNodeSize(TSize(FUNCsize.cx + id->NodeWidth() +
eqsize.cx + fbody->NodeWidth(),
                        fbody->NodeHeight()));
   }
}

void  TFUNCDEC<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{
   if(DoPosFocusFolded(dc, pos)) {
      TSize    FUNCsize, eqsize;
      dc.GetTextExtent("func ", 5, FUNCsize);
      dc.GetTextExtent(" = ",   3, eqsize);
      dc.TextOut(pos, "func");
      int   x = pos.x+FUNCsize.cx;
      TSortID<TFlowViewNode>   *id = GetFuncName();
      id->Draw(dc, TPoint(x, pos.y));
      x += id->NodeWidth(); dc.TextOut(x, pos.y, " = ");
      x += eqsize.cx;
      TSortFBODY<TFlowViewNode> *fbody = GetBody();
      fbody->Draw(dc, TPoint(x, pos.y));
   }
}


void  TID<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TID<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawString(dc, pos, (char *)GetData()); }


void  TINTTYPE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "int"); }
void  TINTTYPE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, "int"); }
void  TCHARTYPE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "char"); }
void  TCHARTYPE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, "char"); }
void  TREALTYPE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "real"); }
void  TREALTYPE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, "real"); }
void  TBOOLTYPE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "bool"); }
void  TBOOLTYPE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, "bool"); }

void  TSTATS<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortSTAT<TFlowViewNode>  *stat;
      int   numchildren = GetNumChildren();
      int   offset;
      int   width = 0, height = 0;
      int   connx = 0;

      for(int i = 0; i<numchildren; i++) {
         stat   = GetStatement(i);
         stat->CalcSize(dc);
         offset = connx-stat->GetConnX();
         if(offset < 0) {
            connx     -= offset;
            width     -= offset;
            offset     = 0;
         }
         if(offset+stat->NodeWidth() > width)
            width = offset+stat->NodeWidth();
         height += stat->NodeHeight() + (i?DistY:0);
      }
      SetConnX(connx);
      SetNodeSize(TSize(width, height));
   }
}

void  TSTATS<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DoPosFocusFolded(dc, pos);
   if(DoPosFocusFolded(dc, pos)) {
      TSize    semicolonsize;
      dc.GetTextExtent(";", 1, semicolonsize);
      int connx = GetConnX();
      int y = pos.y;
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TSortSTAT<TFlowViewNode>  *stat =
GetStatement(i);

         if(i) {
            dc.MoveTo(pos.x+connx, y-DistY);
dc.LineTo(pos.x+connx, y);
            dc.Arrow(TPoint(pos.x+connx, y),
TMyDC::DOWN);
         }
         stat->Draw(dc, TPoint(pos.x+connx - stat-
>GetConnX(), y));
         y += stat->NodeHeight()+DistY;
      }
   }
}

void  TSKIP<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "skip", 0, 0);
}
void  TSKIP<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, "skip", 0, 0);
}

void  TCONCASSIGN<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, " := ", 0);
}
void  TCONCASSIGN<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, 0, " := ", 0);
}


void  TPROCCALL<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, 0, 0);
}
void  TPROCCALL<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, 0, 0);
}

void  TPROCCALLARGS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, "(", ")");
}
void  TPROCCALLARGS<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, 0, "(", ")");
}

void  TIFTHEN<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TFlowViewNode>  *guard = GetGuard();
guard->CalcSize(dc);
      TSortSTATS<TFlowViewNode> *stats =
GetTrueStatements(); stats->CalcSize(dc);

      TSize gsize = guard->NodeSize() + TSize(guard-
>NodeHeight()+TextBorderY*2, TextBorderY*2);
```

```
      SetNodeSize(TSize(max(stats->GetConnX() +
HConnLen*2 + gsize.cx ,
                           stats->NodeWidth() + DistX),
                        gsize.cy+DistY+stats-
>NodeHeight()+VConnLen));
      SetConnX(max(NodeWidth()/2, stats-
>GetConnX()+HConnLen+gsize.cx/2));
   }
}

void  TIFTHEN<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{
   if(DoPosFocusFolded(dc, pos)) {
      TSortEXPR<TFlowViewNode>  *guard = GetGuard();
      TSortSTATS<TFlowViewNode> *stats =
GetTrueStatements();
      int                     d, d2, gx, rx;
      TSize                   gsize;

      d2    = guard->NodeHeight()+TextBorderY*2; d =
d2/2;
      gsize = guard->NodeSize() + TSize(d2,
TextBorderY*2);
      gx    = pos.x + GetConnX() - gsize.cx/2;


      dc.ShadowRectangle(TRect(TPoint(gx, pos.y),gsize),
FillBrush, Pen, ShadowBrush);
      dc.MoveTo(gx+d, pos.y); dc.LineTo(gx, pos.y+d);
dc.LineTo(gx+d, pos.y+d2);
      rx = gx+gsize.cx-1;
      dc.MoveTo(rx-d, pos.y); dc.LineTo(rx, pos.y+d);
dc.LineTo(rx-d, pos.y+d2);

      dc.MoveTo(gx, pos.y+d); dc.LineTo(pos.x+stats-
>GetConnX(), pos.y+d);
      dc.LineTo(pos.x+stats->GetConnX(),
pos.y+gsize.cy+DistY);
      dc.Arrow(TPoint(pos.x+stats->GetConnX(),
pos.y+gsize.cy+DistY), TMyDC::DOWN);


      dc.MoveTo(rx, pos.y+d);
dc.LineTo(pos.x+NodeWidth()-1, pos.y+d);
      dc.LineTo(pos.x+NodeWidth()-1, pos.y+NodeHeight()-
1);
      dc.LineTo(pos.x+stats->GetConnX(),
pos.y+NodeHeight()-1);
      dc.LineTo(pos.x+stats->GetConnX(),
pos.y+NodeHeight()-VConnLen);
      guard->Draw(dc, TPoint(gx+d, pos.y+TextBorderY));
      stats->Draw(dc, TPoint(pos.x,
pos.y+gsize.cy+DistY));
   }
}

void  TIFTHENELSE<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TFlowViewNode>  *guard  = GetGuard();
guard->CalcSize(dc);
      TSortSTATS<TFlowViewNode> *tstats =
GetTrueStatements();  tstats->CalcSize(dc);
      TSortSTATS<TFlowViewNode> *fstats =
GetFalseStatements(); fstats->CalcSize(dc);

      TSize gsize = guard->NodeSize() + TSize(guard-
>NodeHeight()+TextBorderY*2, TextBorderY*2);

      int      t1, t2, f1, f2, d;

      t1 = tstats->GetConnX();  t2 = tstats-
>NodeWidth()-t1;
      f1 = fstats->GetConnX();  f2 = fstats-
>NodeWidth()-f1;

      d = gsize.cx+HConnLen*2-(t2+DistX+f1);
      if(d<0) d = 0;
      SetNodeSize(TSize(t1+t2 + DistX + d + f1+f2,
                        gsize.cy+DistY+max(tstats-
>NodeHeight(),fstats->NodeHeight())+VConnLen));
      SetConnX(t1+(t2+d+DistX+f1)/2);
   }
}

void  TIFTHENELSE<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{
   if(DoPosFocusFolded(dc, pos)) {
      TSortEXPR<TFlowViewNode>  *guard = GetGuard();
      TSortSTATS<TFlowViewNode> *tstats =
GetTrueStatements();
      TSortSTATS<TFlowViewNode> *fstats =
GetFalseStatements();
      int                     d, d2, gx, rx, fx;
      TSize                   gsize;

      d2    = guard->NodeHeight()+TextBorderY*2; d =
d2/2;
      gsize = guard->NodeSize() + TSize(d2,
TextBorderY*2);
      gx    = pos.x + GetConnX() - gsize.cx/2;
      fx    = pos.x+NodeWidth()-fstats->NodeWidth();

      /* Draw Guard */
      dc.ShadowRectangle(TRect(TPoint(gx, pos.y),gsize),
FillBrush, Pen, ShadowBrush);
      dc.MoveTo(gx+d, pos.y); dc.LineTo(gx, pos.y+d);
dc.LineTo(gx+d, pos.y+d2);
      rx = gx+gsize.cx-1;
      dc.MoveTo(rx-d, pos.y); dc.LineTo(rx, pos.y+d);
dc.LineTo(rx-d, pos.y+d2);

      /* Draw TRUE line */
      dc.MoveTo(gx, pos.y+d); dc.LineTo(pos.x+tstats-
>GetConnX(), pos.y+d);
      dc.LineTo(pos.x+tstats->GetConnX(),
pos.y+gsize.cy+DistY);
      dc.Arrow(TPoint(pos.x+tstats->GetConnX(),
pos.y+gsize.cy+DistY), TMyDC::DOWN);

      /* Draw FALSE line */
      dc.MoveTo(rx, pos.y+d); dc.LineTo(fx+fstats-
>GetConnX(), pos.y+d);
      dc.LineTo(fx+fstats->GetConnX(),
pos.y+gsize.cy+DistY);
      dc.Arrow(TPoint(fx+fstats->GetConnX(),
pos.y+gsize.cy+DistY), TMyDC::DOWN);

      dc.MoveTo(fx+fstats->GetConnX(),
pos.y+gsize.cy+DistY+fstats->NodeHeight());
      dc.LineTo(fx+fstats->GetConnX(),
pos.y+NodeHeight()-1);
      dc.LineTo(pos.x+tstats->GetConnX(),
pos.y+NodeHeight()-1);
      dc.LineTo(pos.x+tstats->GetConnX(),
pos.y+gsize.cy+DistY+tstats->NodeHeight());

      guard->Draw(dc, TPoint(gx+d,  pos.y+TextBorderY));
      tstats->Draw(dc, TPoint(pos.x,
pos.y+gsize.cy+DistY));
      fstats->Draw(dc, TPoint(fx,
pos.y+gsize.cy+DistY));
   }
}

void  TWHILE<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortEXPR<TFlowViewNode>  *guard = GetGuard();
guard->CalcSize(dc);
      TSortSTATS<TFlowViewNode> *stats =
GetStatements(); stats->CalcSize(dc);
      TSize                  gsize;

      gsize = guard->NodeSize() + TSize(guard-
>NodeHeight()+TextBorderY*2, TextBorderY*2);

      SetNodeSize(TSize( max(DistX + stats->GetConnX() +
HConnLen*2 + gsize.cx ,
                         stats->NodeWidth() +
DistX*2),

ArrowWidth/2+VConnLen+gsize.cy+DistY+stats-
>NodeHeight()+VConnLen));
      SetConnX(max(NodeWidth()/2, DistX+stats-
>GetConnX()+HConnLen+gsize.cx/2));
   }
}


void  TWHILE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{
   if(DoPosFocusFolded(dc, pos)) {
      int                       d, d2, gx, gy, rx;
      TSize                       gsize;
      TSortEXPR<TFlowViewNode>  *guard = GetGuard();
      TSortSTATS<TFlowViewNode> *stats =
GetStatements();

      d2    = guard->NodeHeight()+TextBorderY*2; d =
d2/2;
      gsize = guard->NodeSize() + TSize(d2,
TextBorderY*2);
      gx    = pos.x + GetConnX() - gsize.cx/2; gy =
pos.y+VConnLen+ArrowWidth/2;

      dc.MoveTo(pos.x+GetConnX(), pos.y);
dc.LineTo(pos.x+GetConnX(), gy);

      /* draw guard surroundings*/
      dc.ShadowRectangle(TRect(TPoint(gx, gy),gsize),
FillBrush, Pen, ShadowBrush);
      dc.MoveTo(gx+d, gy); dc.LineTo(gx, gy+d);
dc.LineTo(gx+d, gy+d2);
      rx = gx+gsize.cx-1;
      dc.MoveTo(rx-d, gy); dc.LineTo(rx, gy+d);
dc.LineTo(rx-d, gy+d2);

      /* draw TRUE line to stats */
      dc.MoveTo(gx, gy+d); dc.LineTo(pos.x+DistX+stats-
>GetConnX(), gy+d);
      dc.LineTo(pos.x+DistX+stats->GetConnX(),
gy+gsize.cy+DistY);
      dc.Arrow(TPoint(pos.x+DistX+stats->GetConnX(),
gy+gsize.cy+DistY), TMyDC::DOWN);

      /* from stats back to top */
      dc.MoveTo(pos.x+DistX+stats->GetConnX(),
pos.y+NodeHeight()-VConnLen);
      dc.LineTo(pos.x+DistX+stats->GetConnX(),
pos.y+NodeHeight()-1);
      dc.LineTo(pos.x,       pos.y+NodeHeight()-1);
      dc.LineTo(pos.x,       pos.y+ArrowWidth/2);
      dc.LineTo(pos.x+GetConnX(), pos.y+ArrowWidth/2);
      dc.Arrow(TPoint(pos.x+DistX+stats-
>GetConnX()+ArrowLength/2,
                      pos.y+ArrowWidth/2),
TMyDC::RIGHT);
      /* draw FALSE line to bottom */
      dc.MoveTo(rx, gy+d);
      dc.LineTo(pos.x+NodeWidth()-1, gy+d);
      dc.LineTo(pos.x+NodeWidth()-1, pos.y+NodeHeight()-
1);
      dc.LineTo(pos.x+GetConnX(),  pos.y+NodeHeight()-
1);

      /* Guard itself */
      guard->Draw(dc, TPoint(gx+d, gy+TextBorderY));
      /* Stat itself */
      stats->Draw(dc, TPoint(pos.x+DistX,
gy+gsize.cy+DistY));
   }
}


void  TREAD<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "read(", ",", ")");
}
void  TREAD<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, "read(", ",", ")");
}

void  TWRITE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "write(", ",", ")");
}
void  TWRITE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "write(", ",", ")");
}

void  TFOR<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortID<TFlowViewNode>    *id     = GetVariable();
         id->CalcSize(dc);
      TSortEXPR<TFlowViewNode>  *from   =
GetFromExpression(); from->CalcSize(dc);
      TSortEXPR<TFlowViewNode>  *by     =
GetByExpression();   by->CalcSize(dc);
      TSortEXPR<TFlowViewNode>  *to     =
GetToExpression();   to->CalcSize(dc);
      TSortSTATS<TFlowViewNode> *stats =
GetStatements();     stats->CalcSize(dc);
      int      maxh, statconnx, connx, owidth,
statwidth;
      TSize    assignsize, testsize, incrsize, plussize;

      dc.GetTextExtent(" := ",   4, assignsize);
      assignsize.cx += id->NodeWidth()+from-
>NodeWidth()+BoxExtraX;
      maxh = max(id->NodeHeight(), from->NodeHeight());
      if(maxh>assignsize.cy)
         assignsize.cy = maxh;
      assignsize.cy += BoxExtraY;

//    gsize = guard->NodeSize() + TSize(guard-
>NodeHeight()+TextBorderY*2, TextBorderY*2);
      dc.GetTextExtent("<=",   2, testsize);
      testsize.cx += id->NodeWidth()+to->NodeWidth();
      maxh = max(id->NodeHeight(), to->NodeHeight());
      if(maxh>testsize.cy)
         testsize.cy = maxh;
      testsize +=
TSize(testsize.cy+TextBorderY*2+RShadW,
TextBorderY*2+BShadH);

      dc.GetTextExtent(" := ", 4, incrsize);
      dc.GetTextExtent("+",    1, plussize);
      incrsize.cx += 2*id->NodeWidth()+by-
>NodeWidth()+plussize.cx+BoxExtraX;
      maxh = max(id->NodeHeight(), by->NodeHeight());
      if(maxh>incrsize.cy)
         incrsize.cy = maxh;
      incrsize.cy += BoxExtraY;


      statconnx = DistX+stats->GetConnX();
      statwidth = DistX*2+max(stats->NodeWidth(),
incrsize.cx);
      if(statconnx < DistX+incrsize.cx/2)
         statconnx = DistX+incrsize.cx/2;
      statwidth = statconnx+max(stats->NodeWidth()-
stats->GetConnX(),
                                 incrsize.cx/2);

      connx = statconnx+HConnLen+testsize.cx/2;

      if(connx < assignsize.cx/2)
         connx = assignsize.cx/2;
      owidth = max(connx+assignsize.cx/2,
connx+testsize.cx/2+HConnLen);



      SetNodeSize(TSize(max(statwidth+DistX, owidth),
                        assignsize.cy + testsize.cy +
                        stats->NodeHeight() +
incrsize.cy + DistY*3 + VConnLen));
      SetConnX(connx);
   }
}

void  TFOR<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(DoPosFocusFolded(dc, pos)) {
      TSortID<TFlowViewNode>   *id    = GetVariable();
      TSortEXPR<TFlowViewNode>    *from  =
GetFromExpression();
      TSortEXPR<TFlowViewNode>    *by    =
GetByExpression();
      TSortEXPR<TFlowViewNode>    *to    =
GetToExpression();
      TSortSTATS<TFlowViewNode>   *stats =
GetStatements();

      int      maxh, x, y, idx, idy, statconnx, gx, gy,
d, d2, rx, ix, iy;
      TSize    assignsize, testsize, incrsize, plussize,
gsize;

      dc.GetTextExtent(" := ",   4, assignsize);
      assignsize.cx += id->NodeWidth()+from-
>NodeWidth()+BoxExtraX;
      maxh = max(id->NodeHeight(), from->NodeHeight());
      if(maxh>assignsize.cy)
         assignsize.cy = maxh;
      assignsize.cy += BoxExtraY;

      x = pos.x+GetConnX()-assignsize.cx/2;
      dc.ShadowRectangle(TRect(TPoint(x,
pos.y),assignsize-TSize(RShadW, BShadH)), FillBrush,
Pen, ShadowBrush);
      idx = x+TextBorderX;
      idy = pos.y+TextBorderY;
      x = idx+id->NodeWidth();
      dc.TextOut(x, idy, " := "); x+=dc.GetTextExtent("
:= ", 4).cx;
      from->Draw(dc, TPoint(x, idy));

      dc.MoveTo(pos.x+GetConnX(), pos.y+assignsize.cy);

//    gsize = guard->NodeSize() + TSize(guard-
>NodeHeight()+TextBorderY*2, TextBorderY*2);
      dc.GetTextExtent("<=",   2, gsize);
      gsize.cx += id->NodeWidth()+to->NodeWidth();
      maxh = max(id->NodeHeight(), to->NodeHeight());
      if(maxh>gsize.cy)
         gsize.cy = maxh;
      gsize += TSize(gsize.cy+TextBorderY*2,
TextBorderY*2);

      d2    = gsize.cy; d = d2/2;
      gx    = pos.x + GetConnX() - (gsize.cx+RShadW)/2;
      gy    = pos.y+assignsize.cy+DistY;
      dc.LineTo(pos.x+GetConnX(), gy);

      dc.ShadowRectangle(TRect(TPoint(gx, gy),gsize),
FillBrush, Pen, ShadowBrush);
      dc.MoveTo(gx+d, gy); dc.LineTo(gx, gy+d);
dc.LineTo(gx+d, gy+d2);
      rx = gx+gsize.cx-1;
      dc.MoveTo(rx-d, gy); dc.LineTo(rx, gy+d);
dc.LineTo(rx-d, gy+d2);

      x = gx+TextBorderX;
      y = gy+TextBorderY;
      id->Draw(dc, TPoint(x, y)); x += id->NodeWidth();
      dc.TextOut(x, y, "<="); x +=
dc.GetTextExtent("<=", 2).cx;

      to->Draw(dc, TPoint(x, y));

      dc.GetTextExtent(" := ", 4, incrsize);
      dc.GetTextExtent("+",    1, plussize);
      incrsize.cx += 2*id->NodeWidth()+by-
>NodeWidth()+plussize.cx+BoxExtraX;
      maxh = max(id->NodeHeight(), by->NodeHeight());
      if(maxh>incrsize.cy)
         incrsize.cy = maxh;
      incrsize.cy += BoxExtraY;

      statconnx = pos.x+DistX+stats->GetConnX();
      if(statconnx < pos.x+DistX+incrsize.cx/2)
         statconnx = pos.x+DistX+incrsize.cx/2;


      /* draw TRUE line to stats */
      dc.MoveTo(gx, gy+d); dc.LineTo(statconnx, gy+d);
      y = gy+gsize.cy+BShadH+DistY;
      dc.LineTo(statconnx, y);
      dc.Arrow(TPoint(statconnx, y), TMyDC::DOWN);

      stats->Draw(dc, TPoint(statconnx-stats-
>GetConnX(), y));
      y+= stats->NodeHeight();

      dc.MoveTo(statconnx, y);
      y+= DistY;
      dc.LineTo(statconnx, y);
      dc.Arrow(TPoint(statconnx, y), TMyDC::DOWN);

      ix = statconnx-incrsize.cx/2; iy = y;
      dc.ShadowRectangle(TRect(TPoint(ix, y),incrsize-
TSize(RShadW, BShadH)), FillBrush, Pen, ShadowBrush);
      x = ix+TextBorderX; y+= TextBorderY;
      id->Draw(dc, TPoint(x, y)); x += id->NodeWidth();
      dc.TextOut(x, y, " := ");   x +=
dc.GetTextExtent(" := ", 4).cx;
      id->Draw(dc, TPoint(x, y)); x += id->NodeWidth();
      dc.TextOut(x, y, "+");      x += plussize.cx;
      by->Draw(dc, TPoint(x, y));
      y = iy+incrsize.cy;
      /* from stats back to top */
      dc.MoveTo(statconnx,        y);
      dc.LineTo(statconnx,        pos.y+NodeHeight()-1);
      dc.LineTo(pos.x,            pos.y+NodeHeight()-1);
      dc.LineTo(pos.x,
pos.y+assignsize.cy+DistY/2);
      dc.LineTo(pos.x+GetConnX(),
pos.y+assignsize.cy+DistY/2);
      dc.Arrow(TPoint(statconnx+ArrowLength/2,
                     pos.y+assignsize.cy+DistY/2),
TMyDC::RIGHT);
      /* draw FALSE line to bottom */
      dc.MoveTo(rx, gy+d);
      dc.LineTo(pos.x+NodeWidth()-1, gy+d);
      dc.LineTo(pos.x+NodeWidth()-1, pos.y+NodeHeight()-
1);
      dc.LineTo(pos.x+GetConnX(),  pos.y+NodeHeight()-
1);

      id->Draw(dc, TPoint(idx, idy));
   }
}

void  TBLOCK<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortDECS<TFlowViewNode>*   decs  =
GetDeclarations(); decs->CalcSize(dc);
      TSortSTATS<TFlowViewNode>*  stats =
GetStatements(); stats->CalcSize(dc);

      SetNodeSize(TSize(2*BlockBorderX+max(decs-
>NodeSize().cx, stats->NodeSize().cx),
                        2*BlockBorderY+decs-
>NodeSize().cy+ArrowLength+stats->NodeSize().cy));
      SetConnX(NodeWidth()/2);
   }
}


void  TBLOCK<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  if(DoPosFocusFolded(dc, pos)) {
      int         y = pos.y+BlockBorderY;

      TSortDECS<TFlowViewNode>*   decs  =
GetDeclarations();
      TSortSTATS<TFlowViewNode>*  stats =
GetStatements();

      dc.SelectObject(ScopePen);
      dc.RectangleOutline(TRect(pos,NodeSize()));
      dc.RestorePen();

      decs->Draw(dc, TPoint(pos.x+BlockBorderX, y));
      y += decs->NodeHeight()+ArrowLength;

      dc.Arrow(TPoint(pos.x+BlockBorderX+stats-
>GetConnX(), y), TMyDC::DOWN);
      stats->Draw(dc, TPoint(pos.x+BlockBorderX, y));
   }
}



void  TVARS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TVARS<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TEXPRS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc,  0, ", ", 0);
}
void  TEXPRS<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TVARVALUE<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      GetVariable()->CalcSize(dc);
SetNodeSize(GetVariable()->NodeSize());
   }
}

void  TVARVALUE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  if(DoPosFocusFolded(dc, pos))
      GetVariable()->Draw(dc, pos);
}
void  TFUNCCALL<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, 0, 0);
}
void  TFUNCCALL<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, 0, 0);
}
```

```
void  TFUNCCALLARGS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, "(", ")");
}
void  TFUNCCALLARGS<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, 0, "(", ")");
}

void  TNUMVALUE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TNUMVALUE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, (char *)GetData()); }

void  TCHARVALUE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TCHARVALUE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, (char *)GetData()); }

void  TIMPLICATION<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " => ", ")");
}
void  TIMPLICATION<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " => ", ")");
}
void  TEQUIVALENCE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " == ", ")");
}
void  TEQUIVALENCE<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " == ", ")");
}

void  TOR<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " \\/ ", ")");
}
void  TOR<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " \\/ ", ")");
}

void  TAND<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " /\\ ", ")");
}
void  TAND<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " /\\ ", ")");
}

void  TLESS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " < ", ")");
}
void  TLESS<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " < ", ")");
}


void  TLESSEQ<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " <= ", ")");
}
void  TLESSEQ<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " <= ", ")");
}


void  TGREATER<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " > ", ")");
}
void  TGREATER<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " > ", ")");
}


void  TGREATEREQ<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " >= ", ")");
}
void  TGREATEREQ<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " >= ", ")");
}


void  TEQUALITY<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " == ", ")");
}
void  TEQUALITY<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " == ", ")");
}

void  TDIFFERENCE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " /= ", ")");
}
void  TDIFFERENCE<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " /= ", ")");
}

void  TADDITION<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " + ", ")");
}
void  TADDITION<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " + ", ")");
}

void  TSUBTRACTION<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " - ", ")");
}
void  TSUBTRACTION<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", " - ", ")");
}

void  TMULTIPLICATION<TFlowViewNode>::CalcSize(TMyDC&
dc)
{  CalcSizeDelimiters(dc, "(", "*", ")");
}
void  TMULTIPLICATION<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "(", "*", ")");
}

void  TDIVISION<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", "/", ")");
}
void  TDIVISION<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", "/", ")");
}

void  TMODULO<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "(", " mod ", ")");
}
void  TMODULO<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "(", " mod ", ")");
}


void  TUNARYMINUS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "-", 0, 0);
}
void  TUNARYMINUS<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "-", 0, 0);
}

void  TNOT<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "NOT ", 0, 0);
}
void  TNOT<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, "NOT ", 0, 0);
}
void  TEXPRBLOCK<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortDECS<TFlowViewNode>*   decs  =
GetDeclarations(); decs->CalcSize(dc);
      TSortSTATS<TFlowViewNode>*  stats =
GetStatements();  stats->CalcSize(dc);
      TSortEXPR<TFlowViewNode>*   expr  =
GetExpression();   expr->CalcSize(dc);

      SetNodeSize(TSize(2*BlockBorderX+max(max(decs-
>NodeSize().cx, stats->NodeSize().cx),expr-
>NodeWidth()),
                        2*BlockBorderY+decs-
>NodeHeight()+ArrowLength+stats-
>NodeHeight()+DistY+expr->NodeHeight()));
      SetConnX(NodeWidth()/2);
   }
}


void  TEXPRBLOCK<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
```

```
{  if(DoPosFocusFolded(dc, pos)) {
      int         y = pos.y+BlockBorderY;

      TSortDECS<TFlowViewNode>*   decs  =
GetDeclarations();
      TSortSTATS<TFlowViewNode>*  stats =
GetStatements();
      TSortEXPR<TFlowViewNode>*   expr  =
GetExpression();

      dc.SelectObject(ScopePen);
      dc.RectangleOutline(TRect(pos, NodeSize()));
      dc.RestorePen();

      decs->Draw(dc, TPoint(pos.x+BlockBorderX, y));
      y += decs->NodeHeight()+ArrowLength;

      dc.Arrow(TPoint(pos.x+BlockBorderX+stats-
>GetConnX(), y), TMyDC::DOWN);
      stats->Draw(dc, TPoint(pos.x+BlockBorderX, y));
      y += stats->NodeHeight()+DistY;
      expr->Draw(dc, TPoint(pos.x+BlockBorderX, y));
   }
}


void  TPBODYSTATS<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortSTAT<TFlowViewNode>  *stat;
      int   numchildren = GetNumChildren();
      int   offset;
      int   width = 0, height = 0;
      int   connx = 0;

      for(int i = 0; i<numchildren; i++) {
         stat   = GetStatement(i);
         stat->CalcSize(dc);
         offset = connx-stat->GetConnX();
         if(offset < 0) {
            connx     -= offset;
            width     -= offset;
            offset     = 0;
         }
         if(offset+stat->NodeWidth() > width)
            width = offset+stat->NodeWidth();
         height += stat->NodeHeight() + (i?DistY:0);
      }
      SetConnX(connx);
      SetNodeSize(TSize(width, height));
   }
}

void  TPBODYSTATS<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DoPosFocusFolded(dc, pos);
   if(DoPosFocusFolded(dc, pos)) {
      int connx = GetConnX();
      int y = pos.y;
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TSortSTAT<TFlowViewNode>  *stat =
GetStatement(i);

         if(i) {
            dc.MoveTo(pos.x+connx, y-DistY);
dc.LineTo(pos.x+connx, y);
            dc.Arrow(TPoint(pos.x+connx, y),
TMyDC::DOWN);
         }
         stat->Draw(dc, TPoint(pos.x+connx - stat-
>GetConnX(), y));
         y += stat->NodeHeight()+DistY;
      }
   }
}

void  TPBODYPARSSTATS<TFlowViewNode>::CalcSize(TMyDC&
dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortPPARS<TFlowViewNode>*  pars  =
GetProcParameters(); pars->CalcSize(dc);
      TSortSTATS<TFlowViewNode>*  stats =
GetStatements();  stats->CalcSize(dc);
```

```
      SetNodeSize(TSize(2*BlockBorderX+max(pars-
>NodeSize().cx, stats->NodeSize().cx),
                        2*BlockBorderY+pars-
>NodeSize().cy+ArrowLength+stats->NodeHeight()));
      SetConnX(NodeWidth()/2);
   }
}


void  TPBODYPARSSTATS<TFlowViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  if(DoPosFocusFolded(dc, pos)) {
      int         y = pos.y+BlockBorderY;
      TSortPPARS<TFlowViewNode>*  pars  =
GetProcParameters();
      TSortSTATS<TFlowViewNode>*  stats =
GetStatements();

      dc.SelectObject(ScopePen);
      dc.RectangleOutline(TRect(pos, NodeSize()));
      dc.RestorePen();

      pars->Draw(dc, TPoint(pos.x+BlockBorderX, y));
      y += pars->NodeHeight()+ArrowLength;

      dc.Arrow(TPoint(pos.x+BlockBorderX+stats-
>GetConnX(), y), TMyDC::DOWN);
      stats->Draw(dc, TPoint(pos.x+BlockBorderX, y));
   }
}


void  TPPARS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TPPARS<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}


void  TPPARVALUE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ": ", 0);
}
void  TPPARVALUE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "? ", ": ", 0);
}

void  TPPARREF<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "! ", ": ", 0);
}
void  TPPARREF<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "! ", ": ", 0);
}

void  TPARGS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TPARGS<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TPARGVALUE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ": ", 0);
}
void  TPARGVALUE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "? ", ": ", 0);
}

void  TPARGREF<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "! ", ": ", 0);
}
void  TPARGREF<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "! ", ": ", 0);
}


void  TFBODY<TFlowViewNode>::CalcSize(TMyDC& dc)
{
   if(GetFolded()) {
      CalcSizeFolded(dc);
   } else {
      TSortFPARS<TFlowViewNode>*  pars  =
GetFuncParameters(); pars->CalcSize(dc);
      TSortEXPR<TFlowViewNode>*   expr  =
GetExpression();     expr->CalcSize(dc);
```

```
      SetNodeSize(TSize(2*BlockBorderX+max(pars-
>NodeSize().cx, expr->NodeSize().cx),
                        2*BlockBorderY+pars-
>NodeSize().cy+DistY+expr->NodeHeight()));
      SetConnX(NodeWidth()/2);
   }
}


void  TFBODY<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  if(DoPosFocusFolded(dc, pos)) {
      int         y = pos.y+BlockBorderY;
      TSortFPARS<TFlowViewNode>*  pars  =
GetFuncParameters();
      TSortEXPR<TFlowViewNode>*   expr  =
GetExpression();

      dc.SelectObject(ScopePen);
      dc.RectangleOutline(TRect(pos, NodeSize()));
      dc.RestorePen();

      pars->Draw(dc, TPoint(pos.x+BlockBorderX, y));
      y += pars->NodeHeight()+DistY;

      expr->Draw(dc, TPoint(pos.x+BlockBorderX, y));
   }
}





void  TFPARS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TFPARS<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TFPARVALUE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ": ", 0);
}
void  TFPARVALUE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "? ", ": ", 0);
}

void  TFARGS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ", ? ", 0);
}
void  TFARGS<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "? ", ", ? ", 0);
}

void  TNUMCASE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "NUMCASE -- flow diagram view not
available");
}

void  TNUMCASE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, "NUMCASE -- flow diagram view not
available");
}

void  TLABCASE<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "LABCASE -- flow diagram view not
available");
}

void  TLABCASE<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, "LABCASE -- flow diagram view not
available");
}

void  TALTS1<TFlowViewNode>::CalcSize(TMyDC&)
{
}

void  TALTS1<TFlowViewNode>::Draw(TMyDC&, TPoint&)
{
}

void  TALTS2<TFlowViewNode>::CalcSize(TMyDC&)
{
}

void  TALTS2<TFlowViewNode>::Draw(TMyDC&, TPoint&)
{
}
void  TALT2<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ": ", 0);
}
void  TALT2<TFlowViewNode>::Draw(TMyDC& dc, TPoint& pos)
{  DrawDelimiters(dc, pos, 0, ": ", 0);
}
void  TLABELS<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ",", 0);
}
void  TLABELS<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ",", 0);
}

void  TNUMLABEL<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TNUMLABEL<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, (char *)GetData()); }

void  TCHARLABEL<TFlowViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TCHARLABEL<TFlowViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, (char *)GetData()); }

typedef TScrolledTreeWindowView<TFlowViewNode>
TFlowView;

DEFINE_RESPONSE_TABLE1(TFlowView, TScrolledWindowView)
   EV_WM_CHAR,
   EV_WM_LBUTTONDOWN,
   EV_WM_RBUTTONDOWN,
   EV_WM_LBUTTONDBLCLK,
   EV_WM_KEYDOWN,
   EV_WM_KEYUP,
   EV_WM_SETFOCUS,
   EV_VN_MIRRORSUBTREE,
   EV_VN_GLOBALFOCUS,

   EV_COMMAND(CM_PREVNODE,       PrevNode),
   EV_COMMAND(CM_NEXTNODE,       NextNode),
   EV_COMMAND(CM_PREVHOLE,       PrevHole),
   EV_COMMAND(CM_NEXTHOLE,       NextHole),
   EV_COMMAND(CM_DELETE,         Delete),
   EV_COMMAND(CM_CUT,            Cut),
   EV_COMMAND(CM_COPY,           Copy),
   EV_COMMAND(CM_PASTE,          Paste),
   EV_COMMAND(CM_PARENT,         Parent),
   EV_COMMAND(CM_FOLD,           ToggleFold),
   EV_COMMAND(CM_ADDHOLETOLIST,  AddHoleToList),
   EV_COMMAND_AND_ID(CM_ADDHOLEBEFORE,
AddHoleBeforeAfter),
   EV_COMMAND_AND_ID(CM_ADDHOLEAFTER,
AddHoleBeforeAfter),
END_RESPONSE_TABLE;

const char far*
TScrolledTreeWindowView<TFlowViewNode>::StaticName()
{
   return "Flow View";
}



DEFINE_DOC_TEMPLATE_CLASS(TStructDocument, TFlowView,
TFlowTemplate);
TFlowTemplate flowTpl("Flow View", "*.STR", 0, "STR",
                      dtAutoDelete | dtUpdateDir
|dtAutoOpen);

const int ImageOffsetX = 8, ImageOffsetY = ArrowLength;

void TFlowView::Paint(TDC& dc, BOOL, TRect&)
{
   dc.SetBkMode(TRANSPARENT);
   dc.SelectObject(Pen);
   dc.SelectObject(FillBrush);
// ((TMyDC&)dc).Arrow(TPoint(ImageOffsetX+root-
>GetConnX(), ImageOffsetY), TMyDC::DOWN);
   TFlowViewNode::FlowTree = &Tree;
   if(Tree.GetRoot()) {
      Tree.GetRoot()->FlushNodePositions();
      Tree.GetRoot()->Draw((TMyDC&)dc,
TPoint(ImageOffsetX, ImageOffsetY));
```

```
   }
}
```

ctxt_pl4.cpp

```
//----------------------------------------------------
----------------------
//    Copyright 1995 Hugo Lyppens
//    Implements class TContextView
//----------------------------------------------------
----------------------
#include "owlhdr.h"
#pragma hdrstop

//#pragma option -Jgx

#include <owl\menu.h>

#include "views.rc"
#include "tree.h"
#include "strucdoc.h"
#include "mydc.h"
#include "viewnode.h"
#include "viewtree.h"
#include "scrwview.h"
#include "scrtrwvw.h"



class   TContextViewNode : public TViewNode {
public:
                           TContextViewNode(int n = 0,
int fixed = FALSE);
   inline TContextViewNode *GetChild(int n) const;
   void                  CalcSizeString(TMyDC&, const
char const *);
   void                  DrawString(TMyDC&, TPoint&,
const char const *);
   void                  CalcSizeHole(TMyDC&);
   void                  DrawHole(TMyDC&, TPoint&);
   void               CalcSizeDelimiters(TMyDC& dc, \
                                             char *start,
char *between, char *end);

   void               DrawDelimiters(TMyDC& dc, TPoint&
pos, \
                                             char *start,
char *between, char *end);
   static TNodeAddress   ActiveAddress;
};

inline TContextViewNode* TContextViewNode::GetChild(int
n) const
{  return (TContextViewNode*)TTreeNode::GetChild(n);
}


const int         Indent  = 20;
const int         DistX   = 4;
const int         DistY   = 4;
#define  FocusColor   TColor(128, 128, 128)
static TBrush            FocusBrush  =
TBrush(FocusColor);

TNodeAddress
TContextViewNode::ActiveAddress;


TContextViewNode::TContextViewNode(int n, int fixed):
   TViewNode(n, fixed)
{
}
void
TContextViewNode::CalcSizeString(TMyDC& dc, const char
const *s)
{
   SetNodeSize(dc.GetTextExtent(s, strlen(s)));
}

void
TContextViewNode::DrawString(TMyDC& dc, TPoint& pos,
const char const *s)
{
   dc.TextOut(pos, s);
}


void
   TContextViewNode::CalcSizeHole(TMyDC& dc)
{
   static char    s[40] = "? <";
   char  *p = GetSortName();
   strcpy(s+3, p); strcpy(s+3+strlen(p), ">");
   CalcSizeString(dc, s);
}

void                 TContextViewNode::DrawHole(TMyDC&
dc, TPoint& pos)
{
   static char    s[40] = "? <";
   char  *p = GetSortName();
   strcpy(s+3, p); strcpy(s+3+strlen(p), ">");
   DrawString(dc, pos, s);
}

void  TPROGHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "No context"); }
void  TPROGHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "No context"); }
void  TDECHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TDECHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TDECSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TDECSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TNDECHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TNDECHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TIDHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TIDHOLE<TContextViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawHole(dc, pos); }
void  TTYPEHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TTYPEHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TSTATHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TSTATHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TSTATSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TSTATSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TVARSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TVARSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TEXPRSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TEXPRSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TEXPRHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TEXPRHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TALTS1HOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALTS1HOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TALTS2HOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALTS2HOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TALT2HOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TALT2HOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TLABELSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TLABELSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TLABELHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TLABELHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPBODYHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPBODYHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPPARSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPPARSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPPARHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPPARHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPARGSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPARGSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TPARGHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TPARGHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TFBODYHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFBODYHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TFPARSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFPARSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TFPARHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFPARHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }
void  TFARGSHOLE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeHole(dc); }
void  TFARGSHOLE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawHole(dc, pos); }


void  TPROG<TContextViewNode>::CalcSize(TMyDC& dc)
{
   TSortDECS<TContextViewNode>    *decs  =
GetDeclarations(); decs->CalcSize(dc);
   TSize                          sz    = decs-
>NodeSize();
   TContextViewNode              *node  = this;
   int                            ind   = Indent;

   for(int i = 0; i<ActiveAddress.Length(); i++) {
      node = node-
>GetChild(ActiveAddress.GetChildNumber(i));
      Sort     sort = node->GetDesiredChildSort(0);
      if(sort==SORT_DECS || sort==SORT_PPARS ||
sort==SORT_FPARS) {
         TContextViewNode           *child = node-
>GetChild(0);
         child->CalcSize(dc);
         if(child->NodeHeight()) {
            sz.cy += child->NodeHeight()+DistY;
            sz.cx  = max(sz.cx,ind+child->NodeWidth());
            ind   += Indent;
         }
      }
   }
   SetNodeSize(sz);
}


void  TPROG<TContextViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{
   TSortDECS<TContextViewNode>    *decs  =
GetDeclarations();
   TPoint                       p = pos;
   TContextViewNode              *node  = this;

   decs->Draw(dc, p); p += TSize(Indent, DistY+decs-
>NodeHeight());
   for(int i = 0; i<ActiveAddress.Length(); i++) {
      node = node-
>GetChild(ActiveAddress.GetChildNumber(i));
      Sort     sort = node->GetDesiredChildSort(0);
      if(sort==SORT_DECS || sort==SORT_PPARS ||
sort==SORT_FPARS) {
         TContextViewNode           *child = node-
>GetChild(0);
         if(child->NodeHeight()) {
            child->Draw(dc, p);
            p += TSize(Indent, DistY+child-
>NodeHeight());
         }
      }
   }
}


void  TDECS<TContextViewNode>::CalcSize(TMyDC& dc)
{
      TSize    semicolonsize;
      dc.GetTextExtent(";",     1, semicolonsize);

      int width       = 0;
      int height      = 0;
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TSortDEC<TContextViewNode> *dec   =
GetDeclaration(i);
         dec->CalcSize(dc);
         int sw = dec->NodeSize().cx+semicolonsize.cx;
         if(sw>width) width = sw;
         height += dec->NodeSize().cy;
      }
      SetNodeSize(TSize(width, height));

}
void  TDECS<TContextViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{
   int y = pos.y;
   int numchildren = GetNumChildren();
   TSize    semicolonsize;
   dc.GetTextExtent(";",     1, semicolonsize);

   for(int i = 0; i<numchildren; i++) {
      TSortDEC<TContextViewNode> *dec   =
GetDeclaration(i);
      if(i) dc.TextOut(pos.x, y, ";");
      dec->Draw(dc, TPoint(pos.x+semicolonsize.cx, y));
      y += dec->NodeSize().cy;
   }
}


void  TContextViewNode::CalcSizeDelimiters(TMyDC& dc,
                                             char *start,
char *between, char *end)
{
      TSize    startsize(0,0), betweensize(0,0),
endsize(0,0);
      if(start)  dc.GetTextExtent(start,
   strlen(start), startsize);
      if(between)dc.GetTextExtent(between,
   strlen(between),  betweensize);
      if(end)    dc.GetTextExtent(end,     strlen(end),
      endsize);

      int width       = startsize.cx;
      int height      = max(max(startsize.cy,
betweensize.cy), endsize.cy);
      int numchildren = GetNumChildren();

      for(int i = 0; i<numchildren; i++) {
         TContextViewNode *child   = GetChild(i);
         child->CalcSize(dc);
         width += child->NodeWidth() +(i<numchildren-
1?betweensize.cx:0);
         if(child->NodeHeight() > height)
            height = child->NodeHeight();
      }
      width += endsize.cx;
      SetNodeSize(TSize(width, height));
}
void  TContextViewNode::DrawDelimiters(TMyDC& dc,
TPoint& pos,
                                             char *start,
char *between, char *end)
{

      TSize    startsize(0,0), betweensize(0,0),
endsize(0,0);
      if(start)  dc.GetTextExtent(start,
   strlen(start), startsize);
      if(between)dc.GetTextExtent(between,
   strlen(between),  betweensize);
      if(end)    dc.GetTextExtent(end,     strlen(end),
      endsize);

      int numchildren = GetNumChildren();

      if(start)dc.TextOut(pos, start);
      int x = pos.x + startsize.cx;
      for(int i = 0; i<numchildren; i++) {
         TContextViewNode *child   = GetChild(i);
         child->Draw(dc, TPoint(x, pos.y));
         x += child->NodeWidth();
         if(i<numchildren-1 && between) {
            dc.TextOut(x, pos.y, between); x +=
betweensize.cx;
         }
      }
      if(end)dc.TextOut(x, pos.y, end);

}


void  TNDECS<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "var  ", ", ", 0);
}
void  TNDECS<TContextViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, "var  ", ", ", 0);
}
void  TNDEC<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc,  0, ":", 0);
}
void  TNDEC<TContextViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ":", 0);
}

void  TPROCDEC<TContextViewNode>::CalcSize(TMyDC& dc)
{
   TSortID<TContextViewNode>    *id = GetProcName();
   id->CalcSize(dc);

   TSize    procsize, eqsize;
   dc.GetTextExtent("proc ", 5, procsize);

   SetNodeSize(TSize(procsize.cx + id->NodeWidth(), id-
>NodeHeight()));
}
void  TPROCDEC<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{
   TSize    procsize, eqsize;
   dc.GetTextExtent("proc ", 5, procsize);
   dc.TextOut(pos, "proc");
   int   x = pos.x+procsize.cx;
   TSortID<TContextViewNode>    *id = GetProcName();
   id->Draw(dc, TPoint(x, pos.y));
}


void  TFUNCDEC<TContextViewNode>::CalcSize(TMyDC& dc)
{
   TSortID<TContextViewNode>    *id = GetFuncName();
   id->CalcSize(dc);

   TSize    FUNCsize;
   dc.GetTextExtent("func ", 5, FUNCsize);

   SetNodeSize(TSize(FUNCsize.cx + id->NodeWidth(),
                  id->NodeHeight()));
}
void  TFUNCDEC<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{
   TSize    FUNCsize;
   dc.GetTextExtent("func ", 5, FUNCsize);
   dc.TextOut(pos, "func");
   int   x = pos.x+FUNCsize.cx;
   TSortID<TContextViewNode>    *id = GetFuncName();
   id->Draw(dc, TPoint(x, pos.y));
}


void  TID<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, (char *)GetData()); }
void  TID<TContextViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawString(dc, pos, (char *)GetData()); }

void  TINTTYPE<TContextViewNode>::CalcSize(TMyDC& dc)
```

```
{  CalcSizeString(dc, "int"); }
void  TINTTYPE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "int"); }
void  TCHARTYPE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "char"); }
void  TCHARTYPE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "char"); }
void  TREALTYPE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "real"); }
void  TREALTYPE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "real"); }
void  TBOOLTYPE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeString(dc, "bool"); }
void  TBOOLTYPE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawString(dc, pos, "bool"); }

void  TSTATS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TSTATS<TContextViewNode>::Draw(TMyDC&, TPoint&){}

void  TSKIP<TContextViewNode>::CalcSize(TMyDC&) {}
void  TSKIP<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TCONCASSIGN<TContextViewNode>::CalcSize(TMyDC&) {}
void  TCONCASSIGN<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TPROCCALL<TContextViewNode>::CalcSize(TMyDC&) {}
void  TPROCCALL<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TPROCCALLARGS<TContextViewNode>::CalcSize(TMyDC&)
{}
void  TPROCCALLARGS<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TIFTHEN<TContextViewNode>::CalcSize(TMyDC&) {}
void  TIFTHEN<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TIFTHENELSE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TIFTHENELSE<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TWHILE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TWHILE<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TREAD<TContextViewNode>::CalcSize(TMyDC&) {}
void  TREAD<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TWRITE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TWRITE<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TFOR<TContextViewNode>::CalcSize(TMyDC&) {}
void  TFOR<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TBLOCK<TContextViewNode>::CalcSize(TMyDC&) {}
void  TBLOCK<TContextViewNode>::Draw(TMyDC&, TPoint&) {}


void  TVARS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TVARS<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TEXPRS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TEXPRS<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TVARVALUE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TVARVALUE<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TFUNCCALL<TContextViewNode>::CalcSize(TMyDC&) {}
void  TFUNCCALL<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TFUNCCALLARGS<TContextViewNode>::CalcSize(TMyDC&)
{}
void  TFUNCCALLARGS<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TNUMVALUE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TNUMVALUE<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TCHARVALUE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TCHARVALUE<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}
void  TIMPLICATION<TContextViewNode>::CalcSize(TMyDC&)
{}
void  TIMPLICATION<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TEQUIVALENCE<TContextViewNode>::CalcSize(TMyDC&)
{}
void  TEQUIVALENCE<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TOR<TContextViewNode>::CalcSize(TMyDC&) {}
void  TOR<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TAND<TContextViewNode>::CalcSize(TMyDC&) {}
void  TAND<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TLESS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TLESS<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TLESSEQ<TContextViewNode>::CalcSize(TMyDC&) {}
void  TLESSEQ<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TGREATER<TContextViewNode>::CalcSize(TMyDC&) {}
void  TGREATER<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TGREATEREQ<TContextViewNode>::CalcSize(TMyDC&) {}
void  TGREATEREQ<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TEQUALITY<TContextViewNode>::CalcSize(TMyDC&) {}
void  TEQUALITY<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TDIFFERENCE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TDIFFERENCE<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TADDITION<TContextViewNode>::CalcSize(TMyDC&) {}
void  TADDITION<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TSUBTRACTION<TContextViewNode>::CalcSize(TMyDC&)
{}
void  TSUBTRACTION<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void
TMULTIPLICATION<TContextViewNode>::CalcSize(TMyDC&) {}
void  TMULTIPLICATION<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TDIVISION<TContextViewNode>::CalcSize(TMyDC&) {}
void  TDIVISION<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TMODULO<TContextViewNode>::CalcSize(TMyDC&) {}
void  TMODULO<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TUNARYMINUS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TUNARYMINUS<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TNOT<TContextViewNode>::CalcSize(TMyDC&) {}
void  TNOT<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TEXPRBLOCK<TContextViewNode>::CalcSize(TMyDC&) {}
void  TEXPRBLOCK<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TPBODYSTATS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TPBODYSTATS<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void
TPBODYPARSSTATS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TPBODYPARSSTATS<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TPPARS<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TPPARS<TContextViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TPPARVALUE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ": ", 0);
}
void  TPPARVALUE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "? ", ": ", 0);
}

void  TPPARREF<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "! ", ": ", 0);
}
void  TPPARREF<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "! ", ": ", 0);
}

void  TPARGS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TPARGS<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TPARGVALUE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TPARGVALUE<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}

void  TPARGREF<TContextViewNode>::CalcSize(TMyDC&) {}
void  TPARGREF<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TFBODY<TContextViewNode>::CalcSize(TMyDC&) {}
void  TFBODY<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TFPARS<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, 0, ", ", 0);
}
void  TFPARS<TContextViewNode>::Draw(TMyDC& dc, TPoint&
pos)
{  DrawDelimiters(dc, pos, 0, ", ", 0);
}

void  TFPARVALUE<TContextViewNode>::CalcSize(TMyDC& dc)
{  CalcSizeDelimiters(dc, "? ", ": ", 0);
}
void  TFPARVALUE<TContextViewNode>::Draw(TMyDC& dc,
TPoint& pos)
{  DrawDelimiters(dc, pos, "? ", ": ", 0);
}

void  TFARGS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TFARGS<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TNUMCASE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TNUMCASE<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TLABCASE<TContextViewNode>::CalcSize(TMyDC&) {}
void  TLABCASE<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TALTS1<TContextViewNode>::CalcSize(TMyDC&) {}
void  TALTS1<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TALTS2<TContextViewNode>::CalcSize(TMyDC&) {}
void  TALTS2<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TALT2<TContextViewNode>::CalcSize(TMyDC&) {}
void  TALT2<TContextViewNode>::Draw(TMyDC&, TPoint&) {}

void  TLABELS<TContextViewNode>::CalcSize(TMyDC&) {}
void  TLABELS<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TNUMLABEL<TContextViewNode>::CalcSize(TMyDC&) {}
void  TNUMLABEL<TContextViewNode>::Draw(TMyDC&, TPoint&)
{}

void  TCHARLABEL<TContextViewNode>::CalcSize(TMyDC&) {}
void  TCHARLABEL<TContextViewNode>::Draw(TMyDC&,
TPoint&) {}


typedef TScrolledTreeWindowView<TContextViewNode>
TContextView;
```

```
DEFINE_RESPONSE_TABLE1(TContextView,
TScrolledWindowView)
   EV_VN_MIRRORSUBTREE,
   EV_VN_NEWACTIVE,
END_RESPONSE_TABLE;

const char far*
TScrolledTreeWindowView<TContextViewNode>::StaticName()
{
   return "Context View";
}

TScrolledTreeWindowView<TContextViewNode>::TScrolledTree
WindowView(TStructDocument& doc, TWindow *parent):
            TScrolledWindowView(doc, parent),
Tree(doc.GetTree())
{
   StructDoc = &doc;
   ShiftDown = FALSE;
}

BOOL
TScrolledTreeWindowView<TContextViewNode>::VnNewActive(T
NodeAddress *)
{
   AdjustScroller();
   Invalidate();
   return FALSE;
}


TSize
TScrolledTreeWindowView<TContextViewNode>::CalcSize()
{
   T   *root = Tree.GetRoot();

   if(root) {
      StructDoc-
>GetActiveAddress(TContextViewNode::ActiveAddress);
      root->CalcSize((TMyDC&)TClientDC(HWindow));
      return root->NodeSize();
   }
   return TSize(0, 0);
}

void TContextView::Paint(TDC& dc, BOOL, TRect&)
{
   dc.SetBkMode(TRANSPARENT);
   if(Tree.GetRoot()) {
      StructDoc-
>GetActiveAddress(TContextViewNode::ActiveAddress);
      Tree.GetRoot()->FlushNodePositions();
      Tree.GetRoot()->Draw((TMyDC&)dc, TPoint(0, 0));
   }
}


DEFINE_DOC_TEMPLATE_CLASS(TStructDocument, TContextView,
TContextTemplate);
TContextTemplate contextTpl("Active Context View",
"*.STR", 0, "STR",
dtAutoDelete | dtUpdateDir |dtAutoOpen);
```

euclides.str

```
PROG[DECS[NDECS[NDEC[ID{x},INT],NDEC[ID{y},INT]]],STATS[
READ[ID{x},ID{y}],IFTHEN[AND[GREATER[VARVALUE[ID{x}],NUM
VALUE{0}],GREATER[VARVALUE[ID{y}],NUMVALUE{0}]],STATS[WR
ITE[VARVALUE[ID{x}],CHARVALUE{''},VARVALUE[ID{y}]],WHILE
[DIFFERENCE[VARVALUE[ID{x}],VARVALUE[ID{y}]],STATS[IFTHE
NELSE[GREATER[VARVALUE[ID{x}],VARVALUE[ID{y}]],STATS[CON
CASSIGN[VARS[ID{x}],EXPRS[SUBTRACTION[VARVALUE[ID{x}],VA
RVALUE[ID{y}]]]]],STATS[CONCASSIGN[VARS[ID{y}],EXPRS[SUB
TRACTION[VARVALUE[ID{y}],VARVALUE[ID{x}]]]]]]]],WRITE[CH
ARVALUE{' '},VARVALUE[ID{x}]]]]]]
```