command `Oplevel-Menu`, bound by default to "`^C Mouse-Right`", is configured in this case to be context sensitive: it presents the user with the usual menu of level-sensitive commands augmented by a submenu of templates (visible in Figure 5.1) appropriate for replacing the placeholder at the cursor.

Pan's other user interaction techniques contribute as well. A standard flag may be added to the panel, as shown in Figure 5.1, that appears when there are placeholders present. This flag (whose default appearance is a purple "`@`" to support the visual theme for syntax-directed editing) indicates that a document is incomplete in the sense being discussed here. A standard highlighter for placeholders may be installed, helping the user locate them at a glance (this highlighter is of course configured by default to color placeholder text in purple).

This approach permits a convenient mix of text-oriented and syntax-directed expansions. For example, once a user sets the level to "`Placeholder`", repeated template expansions may be invoked by iteration over the following sequence of actions:

1. The user points at a placeholder and presses "`Shift-Mouse-Left`"[14] to set the cursor, causing the placeholder to be highlighted and selected textually.

2. The user presses "`Shift-Mouse-Right`" to bring up a context-sensitive menu that includes a submenu of template expansions for the placeholder at the cursor (Figure 5.1). Selecting one invokes the expansion, followed by an automatic reanalysis.

An expansion also leaves the cursor positioned at the first placeholder, if any, in the inserted template; the user may then expand that placeholder in turn with only one additional action. An experienced user would repeat this sequence quickly by holding down the shift key for the duration and invoking commands with mouse buttons only.

A common digression from this purely structural sequence is textual replacement of the placeholder at the cursor, especially for placeholders. This sequence can be fast too:

1. The user types "`^W`" to delete the (already selected) text of the placeholder (the characters "`@identifier`").

2. The user types replacement text.

Unlike placeholder expansion, this sequence leaves the analysis slightly out of date, but the user's next structural move to a placeholder (either by pointing or invoking `Oplevel-Next`) triggers reanalysis (fast in this case since the amount of analysis is kept small by frequent analysis). Other textual digressions are just as convenient, for example moving to the end of the current line, adding a comment, and then moving forward structurally to the next placeholder.

Finally, there is no reason to limit template expansion services in *Pan* to simple expansion of nonterminals in abstract grammars. For example simple clichés of the kind supported by KBEmacs might be inserted (though not necessarily stored in any way to permit their recovery) [135]. Some might be parameterized in complex ways, for example by selection of a program fragment to fill a central "slot" in the cliché; some might need multiple structural selection of the sort advocated in Section 5.3. This all works because template expansion

[14] This keybinding is an *accelerator*, bound for convenience to `Oplevel-Cursor-To-Mouse`, a command whose memorable (and slower) default keybinding uses the `control-C` prefix instead of the shift key.

in *Pan* is implemented on top of kernel mechanisms and language-based infrastructure, not integral to them as with earlier generation syntax-directed editors.

5.6 Incrementality and Inconsistency

Any situation where one kind of information is derived dynamically from another invites **inconsistency**[15] between the two. The syntax-recognizing approach adopted for *Pan*, where language-based information may be derived from document text, is no exception. During text-oriented editing, language-based information maintained by the system sometimes disagrees with what the user sees. For example, *Pan*'s lexical font shifts become incorrect immediately after a statement has been transformed textually into a comment, and no derived information may be available at all for newly entered text.

Inconsistency between text and derived information should not be confused with a related but distinct issue: the well-formedness of a document with respect to its underlying language. Many language-based editors that permit text-oriented editing are able to resolve inconsistency only for well-formed documents: a user who modifies a document textually is not permitted to continue until all changes meet the system's requirements for syntactic well-formedness. *Pan*'s analyzers never fail (Section 4.1), and there is no need for this restriction. It is possible for a document's text to be linguistically well-formed but inconsistent with respect to derived information, for example when the user has just corrected a language error textually. It is equally possible for the document to be ill-formed but consistent, in which case the derived data includes accurate diagnostics. Section 5.4 discusses well-formedness in more detail.

Two general aspects of system behavior concern the problem of inconsistency. The first is how often and under what circumstances the system should reanalyze, given that the mechanisms involved have the potential to confuse the user, to degrade performance, and to make the system's behavior unpredictable. The second is how the system should behave during periods of inconsistency, given that services based on incomplete and unreliable information, when they can work at all, may mislead users. Workable solutions demand delicate compromises involving user interface issues, analysis methods, and system performance.

In *Pan*'s design framework these system issues are refined into four user-centered goals:

1. Don't let analysis obstruct or distract users.
2. Keep the extent of inconsistency small by analyzing often.
3. Make inconsistency visible.
4. Minimize the distinction in service.

Unfortunately these goals conflict. For example, a policy of reanalysis only upon explicit request is nonintrusive, but invites extensive inconsistency. Likewise, making the distinction between the two states visible implies a perceptible threshold, but such a threshold

[15] The term "consistency" is unfortunately used in a variety of ways not directly connected to the meaning intended here. For example, it is sometimes used to name a desirable property of a system's user interface, although it is debatable whether this meaning is either well-defined or useful [49]. Minör uses the term in yet a different way when describing his language-based editor, concerning grammars in trees in this case [87]

can distract when the state changes (for example, after the insertion of a single character immediately following analysis).

Earlier syntax-directed editing systems implicitly addressed subsets of these goals, but they did so by imposing restrictions on users and by not attempting to meet all of them. For example, a pure structure-editor prohibits inconsistency by restricting user actions to syntactically legal transformations on a derivation tree. Hybrid editors bound inconsistency to a structurally-defined "focus" and make it visible, but they impose awkward mode changes on the user, restrict options in the presence of language errors, and may often make the focus much larger than necessary.

The remainder of this section discusses interrelated policies and mechanisms in *Pan*'s design framework that address these goals. These solutions demonstrate that, when the goals are addressed together, workable compromises can be made that are consistent with *Pan*'s design requirements. Fundamental to these solutions is the following assumption:

> An experienced user is almost always a better judge of the extent and implications of recently introduced inconsistency than the system.

After all, the user is responsible for the changes and (mostly) understands them; the system seldom knows what the user is *really* doing.

5.6.1 Unobtrusive Analysis

One way to keep analysis unobtrusive is to make it fast. Pervasive reliance on incremental analysis (Section 4.1) helps. For many simple textual changes, analysis time is roughly proportional to the extent of the changes. Unfortunately, non-local linguistic structure means that the potential for extensive reanalysis is always present. A reasonable assumption here is that experienced users develop an intuition about the kinds of changes that have extensive implications and that they do not perceive variations in analysis time as caprice on the part of the system.

Even well designed incremental analyzers can incur perceptible delays on present-day workstations, so a careful policy is important. An overly ambitious policy, attempting reanalysis after every character insertion or deletion, encounters serious problems. First, analysis at this granularity would find documents nearly always ill-formed. There is little to be gained by insisting that the system perform useful analysis on documents that are in intentionally meaningless states. Second, this drain of resources degrades performance in ways beyond the user's control. Third, over-eager update of the display is visually distracting. The Magpie editor [111] successfully analyzes after every keystroke, but it supports very few services compared to systems like *Pan*.

Two somewhat less ambitious policies are possible. First, the system might *guess* when the document is reasonably well-formed and suitable for analysis. This policy shares the disadvantages of the more ambitious policy—the potential for unpredictable and uncontrollable behavior. Second, the system might restrict text editing to a bounded context (or *focus*) based on some internal structure, performing analysis automatically when the user attempts to leave the context in some way. This policy can create confusion about the exact nature of the context and is incompatible with *Pan*'s general insistence on unrestricted text editing with no overhead on shifts of perspective.

Pan's policy is lazy and predictable, based upon the assumption that the user understands the general state of the document and can judge the tradeoffs involved. Incremental analysis is only performed when requested by the user, either *implicitly* by invoking an operation that triggers automatic reanalysis (Section 4.3.9) or *explicitly* by invoking `Analyze-Changes`.

5.6.2 Frequent Analysis

Nothing in *Pan*'s analysis policy prevents a user from typing an entire document without once invoking the analyzers. This largely defeats the purpose of many *Pan* mechanisms, overtaxes other techniques for dealing with inconsistency, and renders analysis time relatively more obtrusive. Although the user is always the final authority in *Pan*'s design, several techniques encourage frequent analysis.

The overt technique is to deliver helpful applications so that users perceives analysis time as cost-effective. Since invoking strict services causes automatic reanalysis, the more frequently these are used the smaller the increments of analysis. Gracious services do not require consistency, but users will often want to invoke analysis explicitly if services are known to work better as a result.

The covert technique exploits one of *Pan*'s gracious services, lexical font shifting. The font category reserved for unanalyzed text helps the user judge the extent of inconsistency (see below), but it also disturbs the visual harmony of the display during inconsistency, subliminally encouraging frequent analysis to make it more aesthetically pleasing. This only works with appropriate font assignments, as suggested by design guideline 8, which encourages fonts for analyzed ("smart") text that look "nicely typeset," and by design guideline 9, which recommends a font for unanalyzed ("dumb") text that not only looks dumb, thereby reinforcing an important metaphor, but which also "contrasts unpleasantly with other font choices."

5.6.3 Visibility and Soft Thresholds

Inconsistency can have important consequences, so it is important that the user know when this is the case; this is another example of Norman's recommended *mode feedback* [91]. It is also important to give the user whatever help possible in judging the extent of any inconsistency. The cues must be be subtle, however, so that they do not intrude when the user is occupied with tasks for which the information is irrelevant. As a special case, transitions between the two states should be perceptible but not distracting.

It is especially important to avoid threshold effects, distracting visual changes perceived as disproportionate to recent user actions. But a single keystroke in *Pan* can trigger the transition from consistent to inconsistent. The problem is made worse by the fact that this keystroke, the first after an analysis, is nearly always the first of a planned sequence for which the transition is cognitively irrelevant; the user, having made the plan, has already made a commitment to the document becoming inconsistent. In early versions of *Pan* that first keystroke caused all highlighters to be turned off, a severe annoyance to any user making changes by following a highlighted trail through the code. Highlighters no longer do this (see below) and several other techniques help soften the transitions.

The only authoritative indicator of inconsistency is the panel flag "$\wedge$"; this flag, otherwise displayed boldface in dark green, appears dim red during periods of inconsistency. Other visible transitions include color changes for any highlighters configured to be gracious; here careless color choices can produce unpleasant visual thresholds (see below). The extent of inconsistency is revealed in some cases by the amount of raw, unanalyzed text visible, but this cue is not completely reliable: raw text may be present in areas not visible in any window, and there are no visual cues at sites where text has been recently deleted.

It is an open question whether inconsistency might be quantified usefully; if so, revealing inconsistency as a finer-grained change could be helpful to users.

5.6.4 Approximate versus Exact Services

A deeper approach to softening the threshold between consistent and inconsistent states is to make it less important to the user, meaning that the system should behave as much alike in the two states as possible. Maintaining service during inconsistency addresses *Pan*'s fundamental requirement for "uninterrupted service", but risks misleading the user by using obsolete derived data. Presumably because of this risk, other editing systems, and early versions of *Pan* as well, offer no general services that continue to operate in *approximate* modes during inconsistency.

Pan's design framework addresses this prospect with the distinction expressed in the "Strict versus Gracious Services" design metaphor, introduced in Section 4.2.3. A gracious[16] service in *Pan* ideally:

- Operates normally when a document is consistent, using exact information;
- Approximates normal operation otherwise;
- Contributes no distracting behavior to transitions between consistency and inconsistency;
- Operates correctly most of the time when operating in the presence of small inconsistency; and
- Never lies in ways incomprehensible to users.

Notions like "approximate" and "most of the time" are informal but precise, having meaning not in formal language theory but in the cognitive experience of users. By analogy a spelling checker always operates in approximate mode, but well designed ones are both unobtrusive and useful.

Lexer-based font shifting was a gracious service in *Pan* before the concept had been articulated in the design framework. Operationally, the lexer assigns font categories to all text touched during analysis, and those font assignments persist until touched again by the lexer. Applying the "strict" policy (implicitly in effect for all other services at that time) would have meant resetting all fonts to the raw category whenever a document became inconsistent, since the information upon which they were based is suspect. Actually, language theory suggests that only text to the right of any change need be reset this way,

[16] The term derives from the engineering term "graceful degradation," referring to products and services that fail gradually rather than abruptly.

but the visual effects (and loss of almost-correct information) would be just as unpleasant if not more so.

Pan highlighters began as strict services but may now be configured to operate in either mode; design guideline 12 suggests that the choice be made by considering whether approximate operation could be either misleading or useless. To a certain extent this amounts to determining whether local textual changes can have confusing and unpredictable implications for the information being displayed, but there hasn't been enough experience with gracious highlighters to articulate any more precise guidelines.

It is important that colors be assigned carefully to any highlighter configured to work graciously: design guideline 16 suggests that the two relevant colors (one for exact, one for approximate operation) support the "approximate" metaphor by having the same hue, for example, but with a bit less saturation and/or luminance for approximate mode. This avoids distracting thresholds when the state changes, helps makes the state visible, and has the correct connotation for the "quality" of the information being displayed.

Structural navigation in *Pan* remains strict but is a good candidate for elevation to graciousness. *Pan*'s internal tree representation and its connection into the text stream would permit straightforward implementation, but interesting policy questions concerning fine points and boundary questions would have to be explored; these would amount to elaboration of the "location mapping" mechanisms and heuristics described in Section 4.2.7. For example, a textual location in a new region of text would map into nearby structure using simple rules; similarly, a policy would specify a threshold of damage permitted to the textual yield of a tree node to permit placement of the structure cursor on it. As with all gracious services in *Pan*, approximate operation would be an optional convenience: the user can make all services exact at any time with an invocation of `Analyze-Changes`.

5.7 Alternate Views

As noted in Chapter 2, software documents have rich internal structure, with different relationships important to users at different times. In his 1979 essay *Beyond Programming* Winograd argued that multiple, dynamic views of software are important: "The programmer needs to be able to reorganize the information dynamically, looking from one view and then another, going from great generality down to specific detail, and maneuvering around in the space of descriptions to view the interconnections" [139].

Pan's design framework for alternate viewing, described in Section 4.2.8, represents a step in that direction within *Pan*'s current implementation framework. Simple view frameworks can display batches of derived text, permit access to textual annotations on structural components, project component classes onto textual lists, and paint graphical trees with labeled nodes. Many sample views have been implemented within these frameworks; for example tree-to-list projections include a list of error diagnostics, a cross reference of names, a table of contents, and a summary of textual annotations. As with *Pan*'s other elements of user interaction, alternate views will be as useful as the kinds of information used to drive them; the following section describes many possibilities.

Other viewing frameworks would help, giving the view designer the power to filter and transform information in more general ways. Some of these also appear in the following section.

5.8 Other Applications

The ultimate advantage of language-based interaction lies in a rich and open-ended collection of services (organized into thematic or task-related applications) that draw upon a rich repository of information to assist users in commonly performed tasks. This chapter has already described some of these, for example the collection of services that deals with the location and diagnosis of variances (Section 5.4). This final section lists other examples in three categories: those already implemented as prototypes, those unimplemented but straightforward, and those requiring further development or integration with other tools.

In all cases it is emphasized first that applications are implemented by combining generic and language-specific services, and second that the user need not be aware of the complex internal representations needed to make them work. Perceived complexity derives from the particular application and the kind of information involved, not from the elements of user interaction in *Pan*'s design framework through which they can be delivered.

5.8.1 Example Applications

This section describes *Pan* applications that have been prototyped in some way. A few have been mentioned elsewhere, but are included here for completeness.

Structural Search/Replace Like all powerful text editors, *Pan* supports textual replacement based on regular expression matching. However, one sometimes wants replacements to depend on the language structure, not on the textual structure, even when the two are similar. For example, whole-word replacement (where replacing substrings of longer words is not desired) in natural language documents is difficult to specify using patterns. One variant of *Pan*'s replacement command matches patterns only against words (lexemes) as defined by the particular language. Another variant renames variable instances in programs, drawing on information in *Pan*'s database to avoid renaming enclosed variable definitions that have the same lexical name but which are logically different variables.

Queries and Non-Local Navigation One of the few forms of query supported by ordinary text editors is textual search. Searching in *Pan* can draw on *any* derived information. For example, one command locates the declaration and all uses of a program variable, which the user identifies by pointing at any instance. The results of this and other structure-based queries are made available using the query interface described in Section 4.3.8, including a highlighter and operand level for navigation over the results. This particular command is implemented by using a database query defined as part of the *Colander* description in the view style specification. The same underlying *Colander* query supports other services, for example a command that moves the cursor to the declaration of a variable pointed to by the user; this is only one example of a navigation command that follows hypertext-like links defined by the underlying language.

Names, Types, and Values Other sample applications exploit information that is derived in the process of checking a language's static semantics. The cross reference and table of contents view styles (Section 4.3.11) are among these. Another command announces (in the panel) the type of a name, and another announces its value if known. A variant of the latter displays the value temporarily in place of the name. Value computations could be augmented by other sources of information and more advanced reasoning. For example, expressions might be evaluated when enough values are known, and the user might direct that certain assumptions about variable values be made (these might be context- or name-specific "rules," for example to assume that a particular error flag is never true).

Style Checking Stylistic and usage constraints, added to the *Colander* description in a language-based view style, specify violations that do not show up as language errors. Violations can can be viewed by adding a standard panel flag, a standard highlighter, and by navigation at level "`Stylistic Violation`".

Textual Annotations The user can create textual annotations, in a special alternate view for each. A separate summary view lists all existing annotations, and supports shared structural navigation to help the user relate annotation and document component (Section 4.3.11). A programmatic client interface for the facility exists, which might permit a debugger, for example, to annotate various components with context information derived from a stack trace being studied.

Debugger Integration An experimental debugger permits code modifications to be compiled incrementally from *Pan*'s internal representation and patched into an executing image while preserving execution state [18].

5.8.2 Easy Additional Applications

This section describes applications that would not be difficult to implement within *Pan*'s existing framework.

Classes for Library Calls An operand class might be defined to include call sites involving particular libraries, for example calls to window system libraries. Basic services would let users highlight call sites, navigate over them, or generate a separate list view showing all calls. A specialized command available for this operand class might display documentation for a selected library function.

Classes for Particular Language Constructs As part of a software re-engineering task, operand classes might be defined for particular constructs of interest, for example all uses of "goto," allowing rapid determination of their presence, their number, their location, or perhaps the diversity of their target destinations.

Classes for Other Annotations Operand classes might be defined for any other class of information that could be imported from tools in the environment, for example dead code analysis, test coverage, and many others.

Type Checking for Complex Argument Sequences The conventional static-semantic goals for a language, for example ordinary type checking in C, could be extended to check complex, dynamically defined argument sequences to procedures, for example the attribute/value list discipline used by the XView libraries [53]. This argument discipline calls for alternating keywords (defined constants) and values, where the type of the value must be appropriate for the attribute. Some values are sublists, which must be terminated by an argument of zero, and some attributes take no values. None of this can be checked by standard type systems, but they could be encoded into a set of *Colander* constraints specialized for the library.

Current Modifications An operand class might be defined to include all new structure created since some previous reference point (originally visiting the file, or perhaps a user specified reference point). Either a highlighter or a query would make it possible to see code newly introduced during the current session.

5.8.3 Complex Additional Applications

This section describes more speculative *Pan* applications. These are the kind of applications that would exploit more fully *Pan*'s basic design framework, but all would require additional mechanisms and possibly integration with outside tools.

Graphical Cues in Display Small glyphs or icons might be associated with particular flavors (or layers) of structural annotations or properties, and these might appear in the display at each site where one is present. Baecker and Marcus suggest something similar for static document display in their "Essay on Comments," complete with twenty suggestive examples for categories such as "Warning/Sensitive," "Added Code," "Fragile Code," and "Unreachable Code" [6]. This application would require a more flexible presentation engine than *Pan*'s current prototype.

Extra Notation in Display Redundant notation generated by the system can aid comprehension. For example, certain scope markings have been shown helpful for some kinds of reasoning [117] but not for others. Studies such as these focus on language design, in this case examining the tradeoff between redundancy (and therefore more opportunity for entry errors) and comprehension. Generating the extra notation automatically (and optionally) in a system like *Pan* gives the user the best of both choices. This application would require a more flexible presentation engine than *Pan*'s current prototype.

Transformed Notation in Display Notational transformations, produced by the system, can aid comprehension, especially when placed *in situ* in the display so that continuity with surrounding context is not disturbed. These would be helpful for constructs known to be confusing, for example deeply nested conditionals which can be transformed into decision tables or other representations advocated for manual use [99]. This application would require a more flexible presentation engine than *Pan*'s current prototype, as well as some special-purpose analyzers.

Slicing Program slicing is a form of visual program condensation based on the question "what can affect the value of this variable at this point in the program" [137]. A family of queries of this sort might be answered by a general analyzer. This application would require a more flexible presentation engine than *Pan*'s current prototype, as well as integration with a data-flow analyzer.

Control Folding in Display A specialized transformation would elide certain control branches, producing a "collapsed view" based on contextual assumptions. Most helpful would be the assumption that no error-handling branches are taken, yielding a normal-case view to help explain the main program. Identification of error-handling branches might take place by inference, based on pattern matching and stylistic rules, by importation of profile information, by direct annotation from users, or by a mix of these. This application would require a more flexible presentation engine than *Pan*'s current prototype, as well as possible integration with a profiler.

General Debugger Interface *Pan* would serve well as the source code viewing component of a generalized debugger (perhaps multi-language, as is the rest of *Pan*). Special glyphs, mentioned earlier, would mark control and break points in the customary fashion, but *Pan*'s other services would also be available for the user to browse and query. The debugger could be requested to store a stack trace by automatically generated annotations, for example stack frame information at relevant nodes.

Specialized Debugger Interface *Pan* is flexible enough to serve as an interface to many specialized tools, for example a recently developed debugger for optimized code [19]. The debugger displays both the source (the primary view in *Pan*) and the optimized assembly code (an alternate view in *Pan*), with jointly visible stepping (shared navigation in *Pan*). In this debugger certain steps manifest themselves as discontinuous components in the source code because of compiler optimization, a prospect that would be handled by the kind of multi-structural selection described in Section 5.3.4.

Improved Diagnostics A high level diagnostic service, tunable with declarations based on experience, would use as much information as possible (including perhaps judiciously stored history information in the database) to produce better explanations of language errors than is now possible.

Chapter 6

Experience

This chapter reviews some aspects of *Pan*'s design framework in which research results have been mixed and where useful lessons can be learned. The first such aspect is the relationship between *Pan*'s enabling technology (imported and adapted from compiler technology) and the demands of user-centered design. Although considerable progress was made in this area, the legacy of the batch-oriented compiler still causes problems in *Pan*. A second aspect is the difficulty that comes with designing and building a completely new text editor. The cost was extremely high, the result still isn't good enough, and yet no other known editor (neither when the project began nor now) would meet *Pan*'s needs. The third aspect includes a number of observed shortcomings of *Pan*'s mechanisms and user elements for language-based interaction.

6.1 Porting Compiler Technology

From this effort to apply user-centered design have emerged insights concerning the demands that it places on the enabling technology, and in particular on the *Ladle* and *Colander* subsystems in *Pan*. This adds to general evidence, exemplified by Shaw's discussion of input/output mechanisms [115], that underlying computing models designed for batch execution are characteristically unsuited for interactive systems. This section articulates these special demands, explains how advances in the technology (including *Ladle* and *Colander*) have met them partially, and discusses ways in which they remain unsatisfied.

A recurring theme is the failure of the analyzers to capture and exploit the complete *context* of user interaction. For example, incremental algorithms are typically designed to bound the context (and thereby the amount) of computation necessary to process each increment of change by the user [141]. In practice users' changes are related to other user actions separated by space and time. Failure to capture the broader context of interaction causes *loss of information* in surprising (and annoying) ways. Consider for example nesting delimiters `BEGIN` and `END`, which might in an incremental system become unmatched either through addition of a `BEGIN` or deletion of an `END`. Most parsers would produce the same diagnostic in either case, for example "`Unmatched BEGIN`", but to the user who has just accidentally deleted an `END` this message suggests that the system ignores the obvious.[1]

[1] This example is from Wegman and Alberga [136], whose incremental parser would produce different messages in the two cases, for example "`Missing END`" in the latter.

This is a small example, and it has little consequence for users experienced at deciphering messages encoded in the compiler's frame of reference, but there are many more examples like it; each adds a small amount of unnecessary cognitive overhead to the user's task. As with this example, none of the issues raised in this section translate directly into visible breakage in (or absence of) particular services. Rather, each is a potential degradation in overall capacity of the system to provide effective services.

The fact that the issues raised here are *not* directly visible to users is an advantage of the boundary between the lowest two design layers in *Pan*'s implementation model (Figure 3.7 on page 30). This observation renders the issues no less problematical, but it does help characterize a large class of interaction problems. The rest of this section addresses specific challenges to *Pan*'s language-based infrastructure, organized for exposition and not necessarily in order of difficulty or consequence.

6.1.1 Convenient Language Description

Following *Pan*'s requirement for "description-driven support for multiple languages" both language-based analyzers are driven by declarative descriptions, and any number of descriptions may be loaded into a running system (Section 4.1). Writing LALR grammars isn't simple, but *Ladle* eases the burden by permitting primary specification in terms of an abstract syntax (with control over details of the tree representation to be maintained); with only a few additional rules *Ladle*'s preprocessor is able to infer a complete parsing grammar. The *Colander* description language likewise achieves perspicacity with a language modeled on goal satisfaction and named collections of facts [9]; an explicit design goal for that model was to enable descriptions that read naturally, more so for example than attribute grammar specifications [28,105] or action routines [64,84].

Since these goals are common to both batch compilation systems and *Pan*, advances such as these represent mutual progress. Even so, convenient language description remains an elusive goal. For example, *Pan*'s distinction between syntax and contextual constraints (static semantics) is common to almost all language description techniques because of the simplicity it achieves for both description and implementation. But it creates problems in practice for badly designed languages in which, for example, parsing and semantic analysis must be intertwined [35,85], as with the well known "typedef" problem in the C language. Furthermore, experience with *Colander* descriptions suggests that the language, while appropriately expressive, embodies a level of abstraction too low for convenience. Research into general solutions to these problems within *Pan* is currently underway.

Meanwhile, problems that arise in the interactive world confound the picture. For example, there is a need for *description sharing*, both piecewise and in whole, that is poorly addressed by current technology, *Ladle* and *Colander* included. This need arises most simply in *Pan* when two or more different view styles, perhaps written for different users or tasks, are based on identical underlying language descriptions. Slightly more complex examples include the design of view styles with separate concrete grammars but a shared abstract syntax, and the design of view styles with various partial descriptions for static-semantic checking. More complex yet would be the capture of general syntactic and static-semantic abstractions for reuse when describing related languages. Finally, there is a need for descriptions that cross traditional language boundaries, linking together parts of software systems that are realized in more than one language (including documentation languages).

6.1.2 Unconstrainted Text Editing

Pan's requirement for "familiar, unrestricted text editing" demands an unrestricted "smart text editor" that also supports language-based interaction. *Pan* is **syntax-recognizing**, as are Babel [57], the Saga editor [67], and SRE [21]: the user provides text and the system infers the syntactic structure by analysis. An implementation problem with this approach has been the necessity to maintain large and unwieldy syntactic representations for incremental reanalysis. The theory of *grammatical abstraction* [10,22] was developed to resolve this problem in *Pan* (Section 4.1.2).

In contrast, **syntax-directed** systems such the Cornell Program Synthesizer[125], Mentor [29], and Gandalf [50] require that the user explicitly construct internal structure by selecting syntactic templates. The historical distinction between the two approaches reflects divergent goals: users prefer no restrictions on text-based editing, but implementors prefer that modifications be expressed in terms of atomic changes to internal representations.

To users the distinction between the two styles is becoming less sharp. Syntax-directed systems are evolving toward hybrids with increasing text-based support, beginning with Emily [52], which supported almost none, through recent incarnations of the Synthesizer Generator [103], which are increasingly flexible about the context (still structurally defined) in which text editing may take place. Conversely, *Pan* can support a syntax-directed style of editing superimposed on (and coexisting with) the syntax-recognizing model, as described in Section 5.5.

To implementors, however, the distinction continues to reflect design tradeoffs. These don't exist in the world of compilers, where analysis is usually a batch-oriented operation in which history plays no role. The tradeoffs become acute in the presence of incrementality and error recovery, issues to be discussed subsequently.

6.1.3 Incremental Analysis

Incremental computation is conventionally held to be important for interactive program development environments, since useful information must be derived from constantly changing programs. The guiding principle has been that performance gains can be realized by minimizing the amount of computation performed in response to each change [141]. Naive application of this principle to systems like *Pan*, however, ignores the full context of interaction and produces system behavior that is unacceptable to users.

The problem arises from modeling such systems as interactive compilers whose input (program text) is constantly changing and whose behavior depends *only* on the input. This model is inadequate for *Pan*, where users and other applications create ephemeral but useful information superimposed on the linguistic structure of a program. One can think of this information as the state of an ongoing *conversation* between the user and the system. As in any conversation it is important that the referents of the conversation be stable and persistent; incremental analyzers must maintain the *identity* (as opposed to only the equivalence) of program components in the presence of change, a goal not addressed by standard implementations. Systems that support conversations among people encounter similar problems, for example users' "difficulty determining that they were talking about the same objects..." [124].

Ladle's incremental parser, for example, is based on an algorithm that first "unzips" the tree along a path between the leftmost change site and the root; when the algorithm eventually reconstitutes the tree, using new nodes as needed, it has effectively discarded the identity and any annotations associated with nodes along that path. This has serious consequences even from the strictly compiler-oriented point of view. Widely-shared contextual data often appears closer to the root of the tree, and its loss would incur lengthy and often unnecessary recomputation. *Ladle*'s parser ameliorates this effect with a heuristic for reconstituting unzipped nodes: "divided" tree nodes are kept on a special stack and may be reused whenever a node represents the same production in the abstract syntax as in its previous use and when its leftmost child is unchanged between parses. Although effective at restoring tree nodes near the root, this heuristic doesn't always maintain identity in the vicinity of small changes.

Ladle's failure to retain nearby context derives from a deeper problem related to the origin of the technology. This incremental parser is designed as a batch-oriented left-to-right parser with the added ability to be *restarted* at any intermediate point. In the vicinity of a change, the parser examines the new text stream but ignores the *difference* between old and new, which is the real context of the user's modification. This weakness leads to the "`Unmatched BEGIN`" example cited at the beginning of this section.

Earlier generations of syntax-directed editors avoided the problem by permitting textual editing only in the narrowest of contexts, expressions or statements for example [66,126]. But the hybrid approach, which syntax-directed editors have had to adopt for reasons of usability, encounters a variation on the same problem. All textual changes take place in a structurally defined "focus" that amounts to "destructured text"; at reanalysis time it is precisely the local structure that has been discarded and must be reconstrued by the parser.

Because of this weakness, no *Pan* services that depend on identity are robust in the immediate neighborhood of changes, however small. This is a partial failure to meet *Pan*'s requirement for "uninterrupted service".

6.1.4 Granularity of Analysis

In a syntax-recognizing system, the choice of policy for initiating reanalysis has profound implications for the underlying technology. *Pan*'s policy is lazy, only analyzing when the information is needed or when the user requests it explicitly. This implies that derived data and the text stream are often *inconsistent*, which in turn creates problems for user interaction that are described in Section 5.6. It also leads to situations (not anticipated in the design of either *Ladle* or *Colander*) where derived data must be made available, even though it is incomplete and unreliable, during periods of inconsistency.

An opposite extreme, adopted for the Magpie editor [111], is to run the analyzer after every keystroke. This system needs no machinery to deal with inconsistency, but the analyzers must be very fast and extremely adept at dealing with malformed programs (since at this granularity programs are nearly always malformed). Magpie's analyzer is fast, but it drastically curtails analysis upon encountering a single language error and provides fewer services in general. Furthermore this approach exacerbates the problems faced by *Pan*'s analyzers in tracking the identity of program components during change.

The compromise offered by hybrid syntax-directed editors such as the Synthesizer Generator[103] and Gandalf [50] permits textual change only within a structurally defined focus. This bounds the structural context of text-based changes, but typically prohibits navigation away from the focus until its text has been parsed successfully without appearance of language errors; this is an unacceptable interruption of service.

6.1.5 Tolerance for Language Errors

Pan's lazy analysis policy ensures that analyzers are confronted with malformed programs less often than would be the case following other policies. Users tend to request analysis or invoke language-based commands only when they judge their recent changes to be at least locally error-free. Even so, language errors are common and *Pan*'s requirement for "uninterrupted service" applies in their presence. One crucial service in the presence of language errors is the presentation of diagnostics, discussed in Section 5.4. For this and other services to continue operating at all in the presence of errors places pervasive demands on the language-based infrastructure.

Pan's design implicitly extends internal representations to subsume ill-formed programs, and a design goal for *Ladle* and *Colander* was to treat them as interesting but relatively normal occurrences. This happens uniformly, without imposing excessively on language description authors, and it interferes little with other language-based interaction.

Whenever lexical analysis fails, a lexical variance is signaled. For instance, an unterminated comment may lead to a lexical variance. A variance detected during lexical analysis inhibits both parsing and contextual-constraint checking, and all information that existed prior to the attempt to reanalyze the document is preserved.

During parsing *Pan* uses a panic mode mechanism [26] for syntactic error recovery. Directives in *Ladle* descriptions tune the recovery mechanism for each language. The presence of a syntactic variance is marked in the internal tree by an **error subtree** annotated with an appropriate error message. The children of an error node are the lexemes and subtrees that were skipped over during the recovery. This recovery strategy is similar to that used in the Saga editor [67]. Any extant annotations on the subtrees within the error subtree are preserved, including annotations created by *Colander*. Contextual constraints within an error subtree are not attempted by *Colander*, but remain available for possible reuse after correction of the variance. Unsatisfied contextual constraints detected by *Colander* are also variances, resulting in annotations on offending nodes.

Experience suggests that these mechanisms are still inadequate. Panic-based recovery inherits a deficiency, mentioned earlier, of the simple left-right parsing algorithm: it only considers the new text and not the actual change. A more thorough analysis of the changes would permit better heuristics for narrowing the scope of error nodes. Even more problematic, however, the system is unable to restore information (and in particular the identity and annotations of nearby nodes) lost during the presence of even ephemeral and trivial errors. For example, a carelessly typed character adjacent to the outermost `BEGIN` in a program would cause the analyzers to discard nearly all of the useful information about a program; even after an immediate correction (prompted by diagnostic results) expensive and possibly irreplaceable information might be lost. This information loss is an interruption of service, caused by another failure to capture the entire context of user interaction (in this case closely related changes separated by time).

Finally, *Ladle*'s implementation reflects a tendency in the parsing literature to treat error recovery and diagnosis together (to the detriment of both in this case). In *Pan*'s layered implementation model, however, they should be assigned to different layers reflecting different demands being placed upon them. The goal of error *recovery* (part of *Pan*'s infrastructure) is to produce a representation of a badly-formed program with as little loss of information as possible. The goal of error *diagnosis* (a basic user service in the third layer) is to help the user locate, understand, and possibly correct the problem. The two mechanisms are strongly related, both must exploit contextual information, both could be at least partially heuristic, and both must be tuned (perhaps per-language) based on experience. There is no reason, however, to expect that the same mechanisms and configurations will address both goals completely.

6.1.6 Broad Domain of Analysis

Colander's analyzer was designed to be general purpose as well as multi-lingual [9]. Its expressive power suffices for static-semantic checking in programming languages, but it can and has been used for other purposes. For example, an experimental constraint on variable naming conventions required only the addition of a new built-in operator (a regular expression pattern matcher) to the Colander language; since a pattern matcher was already available in the text editor, it was easily added to *Colander* using its internal language extension facilities. But experiments also suggest that finer-grained control over the *Colander* analyzer will be needed as more kinds of analysis are added. One might wish to check language-based constraints frequently while deferring more complex tests. *Colander* analysis at present is monolithic, however; all potentially unsatisfied constraints must be attempted at each invocation.

Experience with *Colander* has revealed other limitations:

- A data flow analyzer implemented in Colander required a convoluted description and resulted in inefficient computation.
- An investigation of stylistic guidelines revealed that non-local structural relationships such as "before" and "after" do not find natural expression in *Colander*, and that some might better be described as tree patterns than as *Colander* goals.
- As described in Section 4.2.6, the *Colander* specification language has no provision for requesting incremental computation of operand class membership and for propagating changes in class membership. General support should be added to the analyzer at a low level.
- Diagnostics are failure-based, so that information *about* unsatisfied constraints is handled specially and is not available uniformly in the database.
- There is no first-class way to deal with partial knowledge in *Colander*. There are no tools to help the author control propagation of failures and spurious diagnostics, so this must be done manually. Similar support for placeholders (Section 5.5) would be helpful.

Whether these would be best addressed by extensions to *Colander* or by additional analyzers (for example, a tree pattern matcher like the one used for prettyprinting in the Mentor system [63,89]) is open pending further research.

6.1.7 Data Sharing

One reason additional analyzers can be considered at all is that *Colander*'s database was designed to be shared with other tools. For example, a prototype context-based prettyprinter attached information to nodes and relied on *Colander* for incremental data propagation [16]. A relatively open data model is crucial for the kinds of user-oriented services that fully exploit the potential for such systems.

Not enough of this sharing was done in *Pan*, however. For historical reasons, *Ladle* and *Colander* use separate data representations, even though much of the structural information ends up being replicated. Aside from the obvious inefficiencies, clients suffer from lack of a uniform access mechanism.

6.2 Building a Text Editor

One of the costliest and perhaps most frustrating parts of the *Pan* project was the creation of an entirely new text editor. It takes a tremendous amount of effort to build and maintain an editor whose textual facilities even approach those of GNU EMACS, which is available without cost. At the time the project began, however, no editor available had the facilities needed, for example an open architecture, a rich internal text representation, and a display engine capable of rendering multiple proportionally spaced fonts using colorful highlighting effects. Section 5.1 discuss *Pan*'s achievements in the effective use of fonts and colors. More recently developed GNU EMACS variants, for example Epoch, have added some facilities of this kind and may be more promising platforms for a system like *Pan*.

6.2.1 Needed Improvements

At the same time, however, *Pan*'s rendering engine remains inadequate; it has constrained the services that could be built as part of this research. All of the following would also be useful:

- Selective elision, including holophrasting [3,119];
- Direct spatial layout, free from the vagaries of explicitly represented whitespace characters (spaces, tabs, and newlines);
- Spatial modes, for example wide margins in which any inserted text is presumed to constitute a comment;
- Graphical cues, including rules (lines), boxes, long arrows, and shading, all used to suggest relationships among textual components [6];
- Special non-textual symbols to annotate text, for example pointing hands, and traffic signs [6]; and
- Nested displays, for example, to permit a program component to be displayed *in situ* using a different (perhaps graphical or tabular) formatting discipline.

Many of these are being addressed as part of the Ensemble project, *Pan*'s successor, by combining high-quality document presentation with *Pan*'s language-based services [48].

6.2.2 Toolkit and UIMS Support

Window system toolkit support for *Pan* was never especially helpful for the most difficult part of the window interface: the text rendering engine. Successive versions of *Pan* were implemented upon Suntools, SunView, and the Athena Widget Set. At each stage text rendering was implemented using raster-level primitives, since nothing better was (or is) available.

More attention is being directed to the development of user interface management systems (UIMS), which permit mappings between internal data, behaviors, and external appearance using declarative specifications [55,58]. It seems unlikely that one would ever support *Pan*'s richly overloaded text display, since it acts as the output channel for so many different kinds of information simultaneously.

6.3 Other Design Issues

Other problems and lessons have become apparent for each of *Pan*'s four design layers, as presented in Figure 3.7 on page 30. The first section in this chapter concerned the infrastructure layer. This section addresses the remaining three.

6.3.1 Kernel

The most glaring omission from *Pan*'s kernel services is the absence of a general mechanism for propagating changes. For historical reasons *Ladle*'s parser, *Colander*'s analyzer, the text representation and rendering engine, scoped editor variables, operand class membership computation, and highlighting all evolved separate mechanisms. Although the interfaces among these components have been carefully controlled as the system evolved, a more general paradigm for event propagation would make their implementations more tractable. One unexplored possibility would be to expand the role of *Colander*'s database and inference engine, to provide low-level support for these functions in a more general way.

6.3.2 Basic Services

Experience has revealed weaknesses in some of the elements of user interaction described in Section 4.3. For example, the structure cursor has been the subject of ongoing development, although the version described in Section 4.3.7 represents a stable point in its evolution. Selection, on the other hand, has proven to be a design problem severe enough to hinder development of some otherwise implementable prototypes. The next evolutionary stage for this aspect of *Pan*'s design, described in Section 5.3.4, would be a mechanism for multiple-component textual selection, with an optional structural aspect in the same sense as the cursor.

Pan's panel flag mechanism has proved useful and robust throughout most applications, but there have been cases where more than two flag states would be appropriate. The flag service was always based on simple boolean operation, so the extension would involve changes at every level from implementation through specification.

6.3.3 View Styles

A number of interesting problems become apparent at the view style layer, where the interface between a class of users and a class of documents must be specified and fine-tuned.

For example, the choice of colors to satisfy various design guidelines has a large impact on the success of *Pan*'s design metaphors, but the decisions can involve subtle differences of colors to get them just right. Unfortunately, color perception is subject to variations in display screens, ambient light, and individual differences, factors that can dominate the subtle color distinctions that are most effective. Convenient personal adjustment is therefore important, but window system support for color specification is crude,[2] and where the *relationships* among colors are paramount, that support is nonexistent. More helpful would be some way for basic color schemes (*sets* of colors in a *Pan* color map, for example) to be continuously adjustable by a single control; thus, a user might dynamically turn the colors up or down on the display, adjusting for local conditions without affecting the basic color assignment scheme. A second control might adjust relative hue selection, much like Tognazzini's "Relative Color" proposal [129].

Font choices are also difficult; as discussed in Section 5.1, none are really suitable. Bigger fonts are much nicer, but take up too much space. Punctuation is especially hard to read in some languages, for example *Colander*, where it is very important to distinguish between commas and periods. Punctuation characters in proportional spaced fonts tuned for text aren't very satisfactory in general for programming languages.

A final meta-problem concerns the fundamental premise of view style design in *Pan*. As the many guidelines in Section 4.4.3 suggest, and the discussions in Chapter 5 confirm, careful configuration is essential to successful implementation of *Pan*'s design metaphors. In *Pan*'s design framework this is considered to be a design task, not necessarily to be carried out by every user. At the same time, however, the requirement for "extensibility and customizability" dictates that the user be permitted to customize freely. Thus any user might, mindfully or not, degrade the effectiveness of *Pan*'s fundamental services with careless or ill-advised customizations. The only practical solution (short of the unacceptable restriction of user rights) rests with plentiful alternative view styles for users to choose among and with presumed close collaboration between the kind of local experts described in Section 4.4.1 and their user population.

[2] *Pan*'s color maps are now specified by selection from a standard X window system list of idiosyncratically named colors.

Chapter 7

Conclusion

This dissertation began with the hypothesis that language-based editing, when implemented with suitable attention to user-centered design principles, can become a practical, usable component in the working environment of experienced software professionals. Investigation of this hypothesis began with a reconsideration of the technology and the assumptions, both stated and unstated, about how it should best be used. From that reconsideration emerged a proposed set of design requirements for such a system.

These requirements have complex implications that often conflict; worse, they threaten to drown users in a sea of system-induced complexity when exactly the opposite effect is needed. The design framework developed as part of this research, applied with a judicious mix of attention to users and tasks, carefully chosen design metaphors, and flexible abstractions, permits such a system to be built.

The *Pan I* Version 4.0 prototype described in this dissertation meets the proposed requirements:

1. *Pan* is a practical, powerful, text editing system whose conventional functionality is never restricted by the addition of language-based features.

2. *Pan* presents documents in a rich text-based information display using multiple, proportionally spaced fonts with potentially many kinds of metainformation (information *about* documents) superimposed via colored highlighting.

3. *Pan*'s interface between user and system is built around a small number of easily understood and flexible services whose apparent complexity derives only from the kind of information being displayed and the task at hand; view styles in *Pan* specify interfaces between users and documents that are tuned for particular tasks and languages.

4. *Pan* supports multiple simultaneous views of software documents, presenting different kinds of information organized in different ways; the framework permits creation of more such views as additional kinds of information become available.

5. *Pan*'s services degrade gracefully when documents contain language-errors, when they are incomplete, and when analysis-derived information is out of date.

6. *Pan* can load any number of declarative language-based view style definitions on demand, each including a *Ladle* description, a *Colander* description, and a specification for user interaction.

7. *Pan* is customizable, extensible, and programmable; it has proven to be an effective, flexible research platform.

The *Pan I* prototype continues to support ongoing research at UC Berkeley and elsewhere; current topics include advanced software viewing and browsing, code optimization and generation, reverse engineering, and static-semantic analysis. Some of *Pan*'s technology is being carried forward into *Pan*'s successor at UC Berkeley, the Ensemble project [48].

Several novel aspects of *Pan*'s design, developed while meeting these goals, deserve recapitulation:

Isolation of Language-Based Technology It is tempting think of language-based editing systems as interactive compilers, but the needs of programmers are dramatically different from those of compilers. Language-based technology developed for compilers ports badly into the domain of user interaction, even in *Pan* where the project began with the benefit of insight from two earlier generations of language-based systems. *Pan*'s layered design model separates language-based analysis mechanisms from user-oriented, language-independent services; most of the system's design accommodates user-centered design choices without excessive coupling to the batch-oriented, compiler model of software structure. *Pan*'s user-oriented services work equally well in an alternate version of the system built with an experimental replacement for the *Colander* analyzer [18].

Operand Class Abstraction The operand class has proven to be an effective abstraction. The description-driven operand class mechanism in *Pan*'s language-independent kernel drives a variety of user-oriented services, ranging from simple navigation to complex projections in alternate views. The abstraction solves several problems in user interface design and permits services to be adapted for uniform operation across multiple language-based view styles.

Gracious Services Metaphor Frequent inconsistency between edited text and analysis-derived data is inescapable in *Pan*, and will persist in any similar system that scales up to confront large-scale propagation of changes. An appropriate design metaphor (as well as some experimental implementations) leads to services that continue to be useful when operating with approximate information.

Elements of User Interaction *Pan* users see a simple system through which an open-ended variety of potentially complex information may be exploited. Simplicity derives from a few basic services that can be applied in a variety of ways, but which have simple and predictable behavior of their own. Examples include highlighters, panel flags, projection views, and the dual-aspect cursor.

Smart Services Metaphor *Pan*'s structure-oriented commands are presented as optional, better-informed elaborations of familiar text-based commands, avoiding the confusion that can arise from a separate command set based on unseen structure.

Coherent Interaction with Document Structure A view style specification describes an interface, implemented by *Pan*, between users and documents in a particular language. Each interface can be tailored for particular users, their tasks, and an underlying language. Much of the richness and effectiveness of *Pan* derives from view style design.

The View Style Designer Formal language description is not an adequate basis for specifying user interaction. A tool like *Pan* embodies a complex relationship among its users, the medium in which they work, and the tasks they perform; it must be *designed* with these factors in mind, a challenging task. The *Pan* system cannot guarantee good design; it offers a framework, building blocks, examples, and guidelines that enable good design.

This research suggests further investigation in several areas.

User Experience More empirical evidence is needed to validate and refine the user interaction techniques developed here. This kind of experience can only be gathered by experimenting with a flexible system such as *Pan* in production environments using production languages.

Advanced Visual Presentation This research suggested many potentially useful presentation techniques based on the static book publishing paradigm. Many of those techniques are not supported by *Pan I*'s prototype rendering engine, but are being explored by the Ensemble project. Just as batch-oriented compiler technology doesn't necessarily port well into an interactive environment, however, some design choices made for a static publishing paradigm may not be appropriate in a more dynamic context, and some techniques may not justify their implementation costs.

Integration with Other Tools *Pan* mechanisms for viewing software documents are designed to exploit a wide variety of possibly large scale information. *Pan*'s potential will only be realized through integration with other tools one expects to find in a modern computer-aided software engineering environment: more ambitious analyzers (data flow for example), debuggers, profilers, test coverage generators, design documentation systems, and persistent storage.

Object-Oriented Programming Much of the experience and insight that drove *Pan*'s design predates widespread acceptance of object-oriented design and languages. These languages are still in flux, and only the most tentative research results are starting to appear that will cast light on the cognitive processes of programmers working in the new design paradigm. Many of *Pan*'s techniques will apply, but new ones will probably be needed to accommodate changing notions of system modularity and interconnectivity.

Language Extension *Pan*'s language description and analysis model is not well suited to languages with powerful extension facilities, for example the macro processing facilities supported by Common Lisp. Closely related is the delivery of services that effectively blur the boundary between language definition and editing system. *Pan*'s techniques for user interaction should apply in most cases, but they may need to be adapted (as the language analysis model must change) for the more dynamic context.

Language-Based Technology Technology that can be shared between language-based editing systems and compilers must be developed and exploited in order to avoid the kinds of infrastructure problems discovered during this research. In the best cases, the boundary between the two applications will become blurred (as it will between editing systems, compilers, and their underlying languages). But it will not succeed until each component of the technology is recast into this new, more general role.

References

[1] James Ambras & Vicky O'Day, "MicroScope: A Knowledge-Based Programming Environment," *IEEE Software* 5 (May 1988), 50–58.

[2] Apple Computers, Inc., *MacWrite Manual*, Cupertino, California, 1984.

[3] J. E. Archer, Jr. & R. Conway, "Display Condensation of Program Text," *IEEE Transactions on Software Engineering* SE-8 (September 1982).

[4] James Archer, Jr. & Richard Conway, "COPE: A Cooperative Programming Environment," Department of Computer Science, Cornell, Technical Report 81-459, June 1981.

[5] Yigal Arens, Lawrence Miller, Stuart C. Shapiro & Norman K. Sondheimer, "Automatic Construction of User-Interface Displays," *Proceedings Seventh National Conference on Artificial Intelligence* (August 1988).

[6] Ronald M. Baecker & Aaron Marcus, *Human Factors and Typography for More Readable Programs*, ACM Press, 1990.

[7] Rolf Bahlke & Gregor Snelting, "The PSG System: From Formal Language Definitions to Interactive Programming Environments," *ACM Transactions on Programming Languages and Systems* 8 (October 1986), 547–576.

[8] Rolf Bahlke & Gregor Snelting, "Design and Structure of a Semantics-Based Programming Environment," *International Journal of Man-Machine Studies* 37 (October 1992), 467–479.

[9] Robert A. Ballance, "Syntactic and Semantic Checking in Language-Based Editing Systems," PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, 89/548, December 1989.

[10] Robert A. Ballance, J. Butcher & Susan L. Graham, "Grammatical Abstraction and Incremental Syntax Analysis in a Language-Based Editor," *Proceedings of the ACM-SIGPLAN 1988 Conference on Programming Language Design and Implementation* 23 (June 22-24, 1988), 185–198.

[11] Robert A. Ballance & Susan L. Graham, "Incremental Consistency Maintenance for Interactive Applications," in *Proceedings of the Eighth International Conference on Logic Programming*, K. Furukawa, ed., MIT Press, Cambridge, Massachusetts, 1991, 895–909.

[12] Robert A. Ballance, Susan L. Graham & Michael L. Van De Vanter, "The Pan Language-Based Editing System," *ACM Transactions on Software Engineering and Methodology* 1 (January 1992), 95–127.

[13] Robert A. Ballance, Michael L. Van De Vanter & Susan L. Graham, "The Architecture of Pan I," Computer Science Division, EECS, University of California, Berkeley, 88/409, August 1987.

[14] Jon Bentley, "Little Languages," *Communications of the ACM* 29 (August 1986), 711–721.

[15] Ted J. Biggerstaff, "Design Recovery for Maintenence and Reuse," *Computer* 22 (July 1989), 36–49.

[16] Christina L. Black, "PPP: The Pan Program Presenter," Computer Science Division, EECS, University of California, Berkeley, 90/589, November 1990, Master's Thesis.

[17] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual, "CENTAUR: the system," *Proceedings ACM SIGSOFT '88: Third Symposium on Software Development Environments* (November 1988).

[18] John Boyland, Charles Farnum & Susan L. Graham, "Attributed Transformational Code Generation for Dynamic Compilers," in *Code Generation – Concepts, Tools, Techniques*, Robert Giegerich & Susan L. Graham, eds., Workshops in Computing Series, Springer Verlag, Berlin, 1992.

[19] Gary Brooks, Gilbert J. Hansen & Steve Simmons, "A New Approach to Debugging Optimized Code," *Proceedings of the ACM-SIGPLAN 1992 Conference on Programming Language Design and Implementation* 27 (June 17-19, 1992), 1–21.

[20] Ruven Brooks, "Using A Behavioral Theory of Program Comprehension in Software Engineering," *Proceedings 3rd International Conference on Software Engineering* (May 1978).

[21] Frank J. Budinsky, Richard C. Holt & Safwat G. Zaky, "SRE – A Syntax Recognizing Editor," *Software–Practice & Experience* 15 (May 1985), 489–497.

[22] Jacob Butcher, "Ladle," Computer Science Division, EECS, University of California, Berkeley, 89/519, November 1989, Master's Thesis.

[23] John M. Carroll & John C. Thomas, "Metaphor and the Cognitive Representation of Computing Systems," *IEEE Transactions on Systems, Man, and Information Sciences* SMC-12 (1982), 107–116.

[24] Ravinder Chandhok, Phillip Miller, John Pane & GlennMeter, "Structure Editing: Evolution Towards Appropriate Use," *Presented at the CHI '90 Workshop on Structure Editors*, Pittsburgh, Pennsylvania (April 1990), Position Paper.

[25] Jacques Cohen, "Constraint Logic Programming Languages," *Communications of the ACM* 33 (July 1990), 52–68.

[26] Robert Paul Corbett, "Static Semantics and Compiler Error Recovery," PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, 85/251, June 1985.

[27] D. D. Cowan, E. W. Mackie & G. M. Pianosi, "Rita–an editor and user interface for manipulating structured documents," *Electronic Publishing* 4 (September 1991), 125–150.

[28] Pierre Deransart, Martin Jourdan & Bernard Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lecture Notes in Computer Science #323, Springer Verlag, Berlin, August 1988.

[29] Veronique Donzeau-Gouge, Gerard Huet, Giles Kahn & Bernard Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments*, David R. Barstow, Howard E. Shrobe & Erik Sandewall, eds., McGraw-Hill, New York, NY, 1984, 128–140, Chapter 7.

[30] Sarah A. Douglas & Thomas P. Moran, "Learning Text Editor Semantics By Analogy," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Boston, Massachusetts (December 1983).

[31] Laura M. Downs & Michael L. Van De Vanter, "Pan I Version 4.0: An Introduction for Users," Computer Science Division, EECS, University of California, Berkeley, 91/659, August 1991.

[32] J. Doyle, "A Truth Maintenance System," in *Readings in Artificial Intelligence*, Bonnie Lynn Webber & Nils J. Nilsson, eds., Tioga, Palo Alto, California, 1981, 496–516.

[33] H. John Durrett, ed., *Color and the Computer*, Academic Press, New York, NY, 1987.

[34] Michael J. Fischer & Richard E. Ladner, "Data Structures for Efficient Implementation of Sticky Pointers in Text Editors," Department of Computer Science, University of Washington, 79-06-08, June 1979.

[35] Bruce T. Forstall, "Programming Language Specification for Editors," Computer Science Division, EECS, University of California, Berkeley, Master's Thesis, November 1991.

[36] Christopher W. Fraser, "Syntax Directed Editing of General Data Structures," *Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16 (June 8-10 1981), 17–21.

[37] Christopher W. Fraser, "A Generalized Text Editor," *Communications of the ACM* 23 (March 1980), 154–158.

[38] Christopher W. Fraser & A. A. Lopez, "Editing Data Structures," *ACM Transactions on Programming Languages and Systems* 3 (April 1981), 115–125.

[39] James A. Galambos, Eloise S. Wikler, John B. Black & Marc M. Sebrechts, "How You Tell Your Computer What You Mean: Ostension in Interactive Systems," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Boston, Massachusetts (December 1983).

[40] E. H. Gansner, J. R. Horgan, D. J. Moore, P. J. Surko, D. E. Swartwout & J. H. Reppy, "SYNED – A Language-Based Editor for an Interactive Programming Environment," *IEEE Computer Society 7th International Computer Software and Applications Conference* (1983).

[41] Michelle Gantt & Bonnie A. Nardi, "Gardeners and Gurus: Patterns of Cooperation Among CAD Users," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Monterey, California (May 3-7, 1992).

[42] David Garlan, "Flexible Unparsing in a Structure Editing Environment," Computer Science Department, Carnegie-Mellon University, CMU-CS-85-129, Pittsburgh, Pennsylvania, April 1985.

[43] David Garlan, "Views for Tools in Integrated Environments," PhD Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-87-147, Pittsburgh, Pennsylvania, May 1987.

[44] Brian Gill-Price, "Personal Communication," 1992.

[45] Adele Goldberg, "Programmer as Reader," *IEEE Software* 4 (September 1987), 62–70.

[46] Dennis R. Goldenson & Marjorie B. Lewis, "Fine Tuning Selection Semantics in a Structure Editor Based Programming Environment: Some Experimental Results," *ACM SIGCHI Bulletin* 20 (October 1988).

[47] John D. Gould, Lizette Alfaro, Rich Finn, Brian Haupt, Angela Minuto & Josiane Salaun, "Why Reading was Slower from CRT Displays than from Paper," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Toronto, Canada (April 1987).

[48] S.L. Graham, M.A. Harrison & E.V. Munson, "The Proteus Presentation System," *Proceedings ACM SIGSOFT '92: Fifth Symposium on Software Development Environments* (December 1992).

[49] Jonathan Grudin, "The Case Against User Interface Consistency," *Communications of the ACM* 32 (October 1989), 1164–1173.

[50] A. Nico Habermann & David Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering* SE-12 (December 1986), 1117–1127.

[51] Heikki Halme & Juha Heinanen, "GNU Emacs as a Dynamically Extensible Programming Environment," *Software–Practice & Experience* 18 (October 1988), 999–1009.

[52] Wilfred J. Hansen, "User Engineering Principles for Interactive Systems," in *Interactive Programming Environments*, David R. Barstow, Howard E. Shrobe & Erik Sandewall, eds., McGraw-Hill, New York, NY, 1984, 217–231, Chapter 11.

[53] Dan Heller, *XView Programming Manual: An OPEN LOOK Toolkit for X11*, O'Reilly & Associates, Inc., Sebastapol, California, 1990.

[54] Paul N. Hilfinger & Phillip Colella, "FIDIL: A Language for Scientific Programming," in *Symbolic Computation: Applications to Scientific Computing*, Robert Grossman, ed., SIAM, 1989, 97–138.

[55] Deborah Hix, "Generations of User-Interface Management Systems," *IEEE Software* 7 (September 1990), 77.

[56] Robert W. Holt, Deborah A. Boehm-Davis & Alan C. Schultz, "Mental Representations of Programs for Student and Professional Programmers," in *Empirical Studies of Programmers: Second Workshop*, Gary M. Olson, Sylvia Sheppard & Elliot Soloway, eds., Ablex Publishing, Norwood, New Jersey, 1987, 33–46.

[57] M. R. Horton, *Design of a Multi-Language Editor with Static Error Detection Capabilities*, PhD Dissertation, Electronics Research Laboratory, EECS, University of California, Berkeley, 1981.

[58] Robert J. K. Jacob, "A Specification Language for Direct-Manipulation User Interfaces," *ACM Transactions on Graphics* 5 (1986), 283–317.

[59] Fahimeh Jalili & Jean H. Gallier, "Building Friendly Parsers," *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico (January 1982).

[60] Vera John-Steiner, *Notebooks of the Mind: Explorations of Thinking*, Harper & Row, 1985.

[61] Jeff Johnson, "Modes in non-computer devices," *International Journal of Man-Machine Studies* 32 (April 1990), 423–438.

[62] Jeff Johnson & George Engelbeck, "Modes Survey Results," *ACM SIGCHI Bulletin* 20 (April 1989), 38–50.

[63] Giles Kahn, Bernard Lang, Bertrand Melese & Elham Morcos, "Metal: A Formalism to Specify Formalisms," *Science of Computer Programming* 3 (1983), 151–188.

[64] Gail E. Kaiser, "Semantics of Structured Editing Environments," PhD Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-85-131, Pittsburgh, Pennsylvania, 17 May 1985.

[65] Gail E. Kaiser, Peter Feiler, Fahimeh Jalili & Johann H. Schlichter, "A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation," *Software–Practice & Experience* 18 (August 1988), 733–748.

[66] Gail E. Kaiser & Elaine Kant, "Incremental Parsing without a Parser," *Journal of Systems and Software* 5 (May 1985), 121–144.

[67] Peter Andre Christopher Kirslis, "The SAGA Editor: A Language-Oriented Editor Based on an Incremental LR(1) Parser," PhD Dissertation, University of Illinois at Urbana-Champaign, UIUCDCS-R-85-1236, December 1985.

[68] Jurgen Koenemann & Scott P. Robertson, "Expert Problem Solving Strategies for Program Comprehension," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, New Orleans, Louisiana (1991).

[69] Glenn E. Krasner & Stephen T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.," *Journal of Object-Oriented Programming* (August/September 1988).

[70] Butler W. Lampson, "Bravo Users Manual," in *ALTO User's Handbook*, Xerox Palo Alto Research Center, Palo Alto, California, 1978.

[71] Bernard Lang, "On the Usefulness of Syntax Directed Editors," in *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen & Dag H. Wanvik, eds., Lecture Notes in Computer Science #244, Springer Verlag, Berlin, 1986, 47–51.

[72] Henry Ledgard, Andrew Singer & John Whiteside, *Directions in Human Factors for Interactive Systems*, Springer Verlag, Berlin, 1981.

[73] Barbara Staudt Lerner, "Automated Customization of Structure Editors," *International Journal of Man-Machine Studies* 37 (October 1992), 529–563.

[74] Stanley Letovsky, "Cognitive Processes in Program Comprehension," in *Empirical Studies of Programmers*, Elliot Soloway & Sitharama Iyengar, eds., Ablex Publishing, Norwood, New Jersey, 1986, 58–79.

[75] Stanley Letovsky & Elliot Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software* 3 (May 1986), 41–49.

[76] Clayton Lewis & Donald A. Norman, "Designing for Error," in *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman & S. W. Draper, eds., Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986, 411–432.

[77] Allan MacLean, Victoria Bellotti, Richard Young & Thomas Moran, "Reaching Through Analogy: A Design Rationale Perspective on Roles of Analogy," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, New Orleans, Louisiana (1991).

[78] Allan MacLean, Kathy Carter, Lennart Lövstrand & Tom Moran, "User-Tailorable Systems: Pressing the Issues with Buttons," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Seattle, Washington (April 1990).

[79] Robert L. Mack, Clayton H. Lewis & John M. Carroll, "Learning to Use Word Processors: Problems and Prospects," *ACM Transactions on Office Information Systems* 1 (July 1983), 254–271.

[80] Wendy E. Mackay, "Patterns of Sharing Customizable Software," *Proceedings Conference on Computer-Supported Cooperative Work*, Los Angeles, California (October 7-10, 1990).

[81] Jock Mackinlay, "Automating the Design of Graphical Presentations of Relational Information," *ACM Transactions on Graphics* 5 (1986), 110–141.

[82] Larry M. Masinter, "Global Program Analysis in an Interactive Environment," Xerox Palo Alto Research Center, SSL-80-1, Palo Alto, California, January 1980.

[83] Raul Medina-Mora, "Syntax Directed Editing: Towards Integrated Programming Environments," PhD Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-81-113, Pittsburgh, Pennsylvania, March 1982.

[84] Raul Medina-Mora & Peter H. Feiler, "An Incremental Programming Environment," *IEEE Transactions on Software Engineering* SE-7 (September 1981), 472–481.

[85] E. A. T. Merks, J. M. Dyck & R. D. Cameron, "Language Design for Program Manipulation," *IEEE Transactions on Software Engineering* 18 (January 1992), 19.

[86] Norman Meyrowitz & Andries van Dam, "Interactive Editing Systems: Parts I and II," *ACM Computing Surveys* 14 (September 1982), 321–415.

[87] Sten Minör, *On Structure-Oriented Editing*, PhD Dissertation, Department of Computer Science, Lund University, Sweden, January 1990.

[88] Sten Minör, "Interacting with Structure-Oriented Editors," *International Journal of Man-Machine Studies* 37 (October 1992), 399–418.

[89] Elham Morcos-Chounet & Alain Conchon, "PPML: A General Formalism to Specify Prettyprinting," *IFIP '86* (April 9, 1986).

[90] Lisa Rubin Neal, "Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Toronto, Canada (April 1987).

[91] Donald A. Norman, "Design Rules Based on Analyses of Human Error," *Communications of the ACM* 26 (April 1983), 254–258.

[92] Donald A. Norman & Stephen W. Draper, eds., *User Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

[93] David Notkin, "Interactive Structure-Oriented Computing," PhD Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-84-103, Pittsburgh, Pennsylvania, February 1984.

[94] David Notkin, "The GANDALF Project," *Journal of Systems and Software* 5 (May 1985), 91–105.

[95] Paul Oman & Curtis R. Cook, "Typographic Style is More than Cosmetic," *Communications of the ACM* 33 (May 1990), 506–520.

[96] Paul W. Oman & Curtis R. Cook, "The Book Paradigm for Improved Maintenance," *IEEE Software* 7 (January 1990), 39.

[97] PROCASE Corporation, *SMARTsystem Reference Guide*, Santa Clara, California, 1990.

[98] John Pasalis, *Realize: An Interactive Graphical Data Structure Presentation System*, Computer Science Division, EECS, University of California, Berkeley, December 1992, Master's Thesis.

[99] G. R. Perkins, R. W. Norman & S. Danicic, "Coping with Deeply Nested Control Structures," *SIGPLAN Notices* 22 (February 1987), 68–77.

[100] Jef Raskin, "Systemic Implications of Leap and an Improved Two-part Cursor: A Case Study," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Austin, Texas (May 1989).

[101] Steven P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 19 (May 1984), 30–41.

[102] Thomas Reps & Tim Teitelbaum, *The Synthesizer Generator Reference Manual*, Department of Computer Science, Cornell University, 1987, Second edition.

[103] Thomas Reps & Tim Teitelbaum, *The Synthesizer Generator Reference Manual*, Springer Verlag, Berlin, 1989, Third edition.

[104] Thomas Reps & Tim Teitelbaum, "The Synthesizer Generator," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 19 (May 1984), 42–48.

[105] Thomas Reps, Tim Teitelbaum & Alan Demers, "Incremental Context Dependent Analysis for Language Based Editors," *ACM Transactions on Programming Languages and Systems* 5 (July 1983), 449–477.

[106] Charles Rich & Richard C. Waters, "The Programmer's Apprentice: A Research Overview," *Computer* 21 (November 1988), 11–25.

[107] Robert S. Rist, "Plans in Programming: Definition, Demonstration, and Development," in *Empirical Studies of Programmers*, Elliot Soloway & Sitharama Iyengar, eds., Ablex Publishing, Norwood, New Jersey, 1986, 28–45.

[108] Caroline Rose, *Inside Macintosh*, Addison Wesley, Reading, Massachusetts, 1985.

[109] Robert W. Scheifler & Jim Gettys, "The X Window System," *ACM Transactions on Graphics* 5 (April 1986), 79–109.

[110] Donald A. Schön, *The reflective practitioner : how professionals think in action*, Basic Books, New York , 1983.

[111] Mayer D. Schwartz, Norman M. Delisle & Vimal S. Begwani, "Incremental Compilation in Magpie," *Proceedings of the ACM-SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices* 19 (June 1984).

[112] Jeffrey Alan Scofield, "Editing as a Paradigm for User Interaction," PhD Dissertation, University of Washington Department of Computer Science, 85-08-10, August 1985.

[113] Andrea A. diSessa, "Models of Computation," in *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman & S. W. Draper, eds., Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986, 201–218.

[114] Mary Shaw, "Beyond Programming-in-the-Large: The Next Challenges for Software Engineering," in *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen & Dag H. Wanvik, eds., Lecture Notes in Computer Science #244, Springer Verlag, Berlin, 1986, 519–535.

[115] Mary Shaw, "An Input-Output Model for Interactive Systems," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Boston, Massachusetts (April 1986).

[116] Louis D. Silverstein, "Human Factors for Color Display Systems: Concepts, Methods, and Research," in *Color and the Computer*, H. John Durrett, ed., 1987, 27–61.

[117] M. E. Sime, T. R. G. Green & D. J. Guest, "Scope Marking in Computer Conditionals – A Psychological Evaluation," *International Journal of Man-Machine Studies* 9 (1977), 107–118.

[118] Barbara Smith & Gerald Kelleher, eds., *Reason Maintenance Systems and Their Applications*, Series in Artificial Intelligence, Ellis Norwood Limited, Chichester, 1988.

[119] Scott R. Smith, David T. Barnard & Ian A. Macleod, "Holophrasted Displays in an Interactive Environment," *International Journal of Man-Machine Studies* 20 (April 1984), 343–355.

[120] Elliot Soloway & Kate Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering* SE-10 (September 1984), 595–609.

[121] Richard Stallman, "GNU Emacs Manual," Fifth Edition, Emacs Version 18 for Unix Users, October 1986.

[122] Richard M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," *Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16 (June 8-10 1981), 147–156.

[123] Ola Strömfors, "Editing Large Programs Using a Structure-Oriented Editor," in *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen & Dag H. Wanvik, eds., Lecture Notes in Computer Science #244, Springer Verlag, Berlin, 1986, 39–46.

[124] Deborah G. Tatar, Gregg Foster & Daniel G. Bobrow, "Design for Conversation: Lessons from Cognoter," *International Journal of Man-Machine Studies* 34 (1991), 185–209.

[125] Tim Teitelbaum & Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM* 24 (September 1981), 563–573.

[126] Tim Teitelbaum, Thomas Reps & Susan Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer," *Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16 (June 8-10 1981), 8–16.

[127] Warren Teitelman, "A Tour Through Cedar," *IEEE Transactions on Software Engineering* SE-11 (March 1985), 285–302.

[128] Larry Tesler, "The Smalltalk Environment," *BYTE* 6 (August 1981), 90–147.

[129] Bruce Tognazzini, *Tog on Interface*, Addison Wesley, Reading, Massachusetts, 1992.

[130] Michael L. Van De Vanter, "Debugging and Error Management in Pan I," Computer Science Division, EECS, University of California, Berkeley, 89/554, December 1989.

[131] Michael L. Van De Vanter, "The Text Module in Pan I," Computer Science Division, EECS, University of California, Berkeley, PIPER Working Paper 90-2, May 1991.

[132] Michael L. Van De Vanter, Robert A. Ballance & Susan L. Graham, "Coherent User Interfaces for Language-Based Editing Systems," *International Journal of Man-Machine Studies* 37 (October 1992), 431–466.

[133] Neff Walker & Judith Reitman Olson, "Designing Keybindings to be Easy to Learn and Resistant to Forgetting Even When the Set of Commands is Large," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Washington, DC (May 1988).

[134] R. C. Waters, "Program editors should not abandon text oriented commands," *SIGPLAN Notices* 17 (1982), 39–46.

[135] Richard C. Waters, "The programmers apprentice: a session with KBEmacs," *IEEE Transactions on Software Engineering* SE-11 (November 1985), 1296–1320.

[136] Mark N. Wegman & Cyril N. Alberga, "Parsing for structural editors (part II)," IBM Corporation, Research Division, RC 9197, Yorktown Heights, NY, January 6, 1982, Extended abstract.

[137] Mark Weiser, "Program Slicing," *IEEE Transactions on Software Engineering* SE-10 (July 1984), 352–357.

[138] Jim Welsh & Mark Toleman, "Conceptual Issues in Language-Based Editor Design," *International Journal of Man-Machine Studies* 37 (October 1992), 419–430.

[139] Terry Winograd, "Beyond Programming Languages," *Communications of the ACM* 22 (July 1979), 391–401.

[140] Steven R. Wood, "Z–The 95 Percent Program Editor," *Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16 (June 8-10 1981), 1–7.

[141] Daniel M. Yellin & Robert E. Strom, "INC: A Language for Incremental Computations," *ACM Transactions on Programming Languages and Systems* 13 (April 1991), 211.

Appendix A

Annotated View Style Specification for Modula2

;;; Create a new view style called "modula2," to be based on the standard Ladle *and*
;;; Colander *specifications for modula2 (not shown). This view style specification is*
;;; written in a provisional language described more fully in Appendices B and C.

```
(lang:Define-Language-View-Style "modula2"
```

;; Add to the new view style a default configuration for many language-based features:
;; menus and key bindings for language-based operations, standard panel flags, and default
;; fonts and colors. This keeps the specification small and promotes a uniform look and feel
;; across view styles. All of the standard text-based behavior and configuration is
;; inherited unchanged from the global configuration context.

```
  (:add-defaults)
```

;; Build up the basic menu of services, named after this view style, by combining standard
;; parts, following conventions for all view styles. Add extra features, for example
;; automatically generated menus for controlling highlighters and operand levels, as well as
;; special views that are supported by the underlying Colander *specification.*

```
  (:default-menu
    (:menu "Modula2"
          (user::global-clipboard-menu)
          (user::global-textedit-menu)
          (user::global-undo-menu)
          (user::global-store-menu)
          ((:copy-menu user::global-language-menu "Language"
                      ((:level-menu) "Levels")
                      ((:highlighter-menu) "Highlighters")))
          ((:copy-menu user::global-names-menu "Modula2 Names"
                      (lang:Visit-Cross-Ref-View "Visit Cross Ref. View")))
          ((:copy-menu user::language-view-menu "View"
                      (lang:Visit-Cross-Ref-View "Names Cross Reference" 3)
                      (lang:Visit-TOC-View "Proc. Table of Contents" 4)))
          (user::global-text-window-menu "Window")
          (user::global-lang-help-menu)
          ))
```

```
(:define-operand-classes

  ;; Some operand classes, for example "Language Error," are predefined and can be used in
  ;; any view style. Define here a basic set of view-style-specific operand classes to drive
  ;; standard services. The first few are purely syntactic; each is defined by a set of tree
  ;; operator names for the Ladle internal representation, taken from the associated
  ;; Ladle specification for modula2.

  (modula2-proc-opclass
    :title              "Procedure"
    :documentation      "Modula2 procedures."

    ;; For this view style, don't distinguish among procedures, functions, and modules.

    :definition         (:operators "procedure_declaration"
                                    "function_declaration"
                                    "module_declaration")

    ;; Declare that this is a "structural" operand class (the ordinary case), and that all
    ;; the default language-based commands should be made available through the operand
    ;; level command dispatch mechanism.

    :bind-structural-commands

    ;; Use a standard operand menu available that's titled "Procedure"

    :operand-menu       user::procedure-oplevel-menu
    :before             lang:Analyze-Quietly)

  (modula2-decl-opclass
    :title              "Declaration"
    :documentation      "Modula2 declarations."
    :definition         (:operators "declaration"
                                    "definition")
    :bind-structural-commands
    :operand-menu       user::declaration-oplevel-menu
    :before             lang:Analyze-Quietly)

  (modula2-stmt-opclass
    :title              "Statement"
    :documentation      "Modula2 statements."
    :definition         (:operators "stmt")
    :bind-structural-commands
    :operand-menu       user::statement-oplevel-menu
    :before             lang:Analyze-Quietly)

  (modula2-expr-opclass
    :title              "Expression"
    :documentation      "Modula2 expressions."
    :definition         (:operators "expr")
    :bind-structural-commands
    :operand-menu       user::expression-oplevel-menu
    :before             lang:Analyze-Quietly)
```

```
;; The next few operand class definitions serve to parameterize standard services that deal
;; with named entities in programming languages. For example, the first specifies what
;; structural components correspond to uses of names in the language. Although they could be
;; used to define operand levels, for supporting language-based navigation and the like, they
;; are not judged to be useful for that purpose.

  (modula2-id-opclass
    :title              "Identifier Instance"
    :documentation      "Lexemes - occurrences of identifiers."
    :definition         (:operators "id")
    :set-command        nil
    :before             lang:Analyze-Quietly)

;; Class definition to support Identifier Cross Reference.

  (modula2-decl-id-opclass
    :title              "Id Declaration"
    :documentation      "Lexemes - declarations of identifiers."
    :definition         (:and
                          (:operators "id")
                          (:grandparent-operators "var_declaration"))
    :set-command        nil
    :before             lang:Analyze-Quietly)

;; Class definition to support Procedure Table of Contents.

  (modula2-header-opclass
    :title              "Procedure/Function Header"
    :documentation      "Procedure name and argument list"
    :definition         (:operators "procedure_heading"
                                    "function_heading")
    :set-command        nil
    :before             lang:Analyze-Quietly)
  )

;; Parameterize various language-based services for this view style, using view-style-specific
;; operand class definitions (see above).

(:option-values
  lang:Name-Instance-Opclass-Name      modula2-id-opclass
  lang:Name-Decl-Opclass-Name          modula2-decl-id-opclass
  lang:TOC-Header-Opclass-Name         modula2-header-opclass)
```

```
;; Define the menu of operand levels available to users in this view style, specified in terms
;; of both generic and view-style-specific operand class definitions. Add keystroke bindings as
;; accelerators for selecting them quickly, following general and view-style specific conventions.

(:operand-levels      (text:character-level   "^c c")
                      (text:word-level        "^c w")
                      (text:line-level        "^c l")
                      (lang:note-opclass      "^c n")
                      (lang:lexeme-opclass    "^c x")
                      (modula2-expr-opclass   "^C e")
                      (modula2-stmt-opclass   "^C s")
                      (modula2-decl-opclass   "^C d")
                      (modula2-proc-opclass   "^C p")
                      (lang:query-opclass     "^C q")
                      (lang:lang-err-opclass  "^C !" "^C #"))

;; Some highlighters, for example one for the class "Language Error," are predefined and
;; can be used in any view style. Define here view-style-specific highlighters.

(:define-highlighters

  ;; Create a highlighter for procedure headers that uses the color specified in the
  ;; third slot of the foreground color map for the view style. This highlighter is gracious
  ;; (not strict), meaning that it continues to operate (in approximate mode) during periods
  ;; when text and derived structure are inconsistent; during inconsistent periods, however,
  ;; the highlighter display color is taken from the alternate foreground color map for the
  ;; view style, which should by convention be a similar but less saturated version of its
  ;; counterpart in the primary color map.

  (modula2-header-highlighter
    :opclass          modula2-header-opclass
    :title "Function & Procedure Headers"
    :documentation    "An example using color to set off procedures."
    :apropos          (function procedure)
    :effect           :fg3
    :strict?          nil))

;; Define the collection of highlighters available to users in this view style, combining
;; some generic predefined ones with the view-style-specific one defined above. Specify
;; whether each is to be initially on or off. Add keystroke bindings as accelerators for
;; toggling them quickly, following general and view-style specific conventions. Do not
;; permit users to turn off highlighting of the structure cursor in this view style.

(:highlighters        (lang:lang-err-highlighter  :on   "^C ^C !")
                      (lang:query-highlighter     :on   "^C ^C q")
                      (modula2-header-highlighter :on   "^C ^C p")
                      (lang:note-highlighter      :off  "^C ^C n")
                      (lang:cursor-highlighter    :on))

;; Configure the appearance of window panels for the view style, beginning with.
;; a view style logo.

(:option-values
  win:Panel-View-Logo-Bitmap           "mod2-logo.29x17")
```

```
;; Add panel flags to the default set inherited from the global configuration. The default
;; set includes one that appears when the view permits no editing, one that indicates unsaved
;; modifications, and one that announces when automatic text filling is in effect. The flags
;; added here are specified using scoped variables (whose local values dynamically define
;; whether the flag is on or off), and standard bitmaps and colors for a uniform look and
;; feel across view styles.

(:add-flags
  (user:%pad-flag-variable%          user::Space-Bitmap)
  (otree:Parse-Is-Current?            (user::Parse-Current-Bitmap
                                       user::Panel-Flag-Green)
                                      (user::Parse-Not-Current-Bitmap
                                       user::Panel-Flag-Pale-Red))
  (sem:Database-Updated?              (user::Database-Updated-Bitmap
                                       user::Panel-Flag-Green)
                                      (user::Database-Not-Updated-Bitmap
                                       user::Panel-Flag-Pale-Red))
  (lang:Language-Errors-Present?      (user::Language-Errors-Present-Bitmap
                                       user::Panel-Flag-Red)))

;; Configure text rendering for the view style.

(:option-values

  ;; Assign font selections to the font map for the view style. There are eight slots but only
  ;; five are used by the current implementation. Font assignments are made by the lexer as
  ;; follows: (0) newly entered, unanalyzed text; (1) scanned but not parsed (this slot is only
  ;; effective during debugging); (2) language keywords; (3) comments; and (4) language
  ;; identifiers.

  twin:Text-Window-Fontmap
               ("-*-lucidatypewriter-medium-r-*-*-12-*-*-*-*-*-*-*"
                "-*-lucida-medium-r-*-*-12-*-*-*-*-*-*-*"
                "-*-lucidabright-medium-r-*-*-12-*-*-*-*-*-*-*"
                "-*-new century schoolbook-medium-r-*-*-12-*-*-*-*-*-*-*"
                "-*-lucidabright-demibold-r-*-*-12-*-*-*-*-*-*-*")

  ;; Assign font selections to the alternate font map for the view style. A single, global
  ;; "demo" switch (option variable) controls whether the demo fontmaps are used.

  twin:Text-Window-Demo-Fontmap
               ("-*-lucidatypewriter-medium-r-*-*-17-*-*-*-*-*-*-*"
                "-*-lucida-medium-r-*-*-18-*-*-*-*-*-*-*"
                "-*-lucidabright-medium-r-*-*-18-*-*-*-*-*-*-*"
                "-*-lucidabright-medium-r-*-*-14-*-*-*-*-*-*-*"
                "-*-lucidabright-demibold-r-*-*-18-*-*-*-*-*-*-*")

  ;; Specify default (initial) size for newly created text windows in the view style, computed
  ;; relative to the nominal width and height of the default (font map slot zero) font for the
  ;; view style.

  twin:Text-Window-Init-Cols          60
  twin:Text-Window-Init-Rows          25
```

```
;; Render text using a tabwidth computed in pixels as 5 times the nominal width of the
;; default (font map slot zero) font for the view style. This width is used for all fonts in
;; the view style so tab alignment may be obtained in the presence of multiple,
;; proportionally-spaced fonts.

twin:Text-Window-Tabwidth           5

;; Assign color selections to the foreground (ink) color map for the view style. There are
;; four slots corresponding to possible ink selections, with the default (the first) color
;; being black by convention. Each slot contains a pair of color specifications: the left
;; side is used for color monitors and the right side for monochrome monitors. These color
;; specifications are normally inherited unchanged from the global defaults in order to
;; promote a uniform look and feel.

twin:Text-Window-Fg-Colormap        (("black" . "black")
                                     ("IndianRed1" . "black")
                                     ("SlateBlue3" . "black")
                                     ("turquoise3" . "black"))

;; Assign color selections to the background (shading) color map for the view style. There
;; are three slots corresponding to three possible shadings, where the background is assumed
;; to be white when no shading is in effect.

twin:Text-Window-Bg-Colormap        (("LightYellow1" . "white")
                                     ("alice blue" . "white")
                                     ("lavender" . "white"))

;; Assign color selection to alternate versions of the two color maps: foreground and
;; background. These alternate versions are used during periods when text and derived
;; language-based information are inconsistent. User interface design convention dictates
;; that these colors be chosen similar in hue to their counterparts in primary color maps,
;; but with reduced saturation to suggest metaphorically that the information they represent
;; is not strictly reliable.

twin:Text-Window-Fg-Alt-Colormap    (("black" . "black")
                                     ("IndianRed3" . "black")
                                     ("SkyBlue2" . "black")
                                     ("turquoise4" . "black"))

twin:Text-Window-Bg-Alt-Colormap    (("ivory" . "white")
                                     ("azure" . "white")
                                     ("lavender blush" . "white"))

;; There is also "demo" version for each of the above four color maps (now shown here).
;; Color assignments suitable for the single user don't show up at a distance, so demo colors
;; are typically brighter and more saturated.

))

;;; This concludes the view style specification for the view style named "modula2." Any of
;;; the many other configuration options in Pan may also be assigned values in a view
;;; style specification. Templates may be specified to add syntax-directed style of editing
;;; to standard language-based commands. New view-style-specific commands may be defined and
;;; bound to keystrokes or menus. Finally, procedures may be attached to various "hooks"
;;; run at predefined events (for example at view initialization and after each analysis).
```

Appendix B

Language-Based View Style Specification

A language-based view style in *Pan* is a design for interaction between some intended category of users, performing some collection of tasks, and their software documents expressed in some formal language notation. This appendix summarizes how such a design can be recorded in *Pan*'s provisional specification language.

Language-based view styles are constructed in isolation from the main *Pan* system and from each other. The system has (almost) no information concerning the name, the location, or even the existence of any view style until it is loaded. View styles (and in particular the *Ladle* and *Colander* descriptions they use) can be loaded at any time, as many as desired, as long as they do not conflict in name. Each contains all specifications necessary to enable the system to provide a wide variety of language-based services in the target language, including per-language control of almost every aspect of user interaction. A sample specification for Modula-2 appears in Appendix A.

B.1 Associating View Styles with File Types

View style configuration in *Pan* is arranged so that the system contains as little information as possible about view styles not yet loaded. Something must be known, however, so that a view style can be located and loaded when needed. A declaration of the following form supplies the needed information for *Pan*'s file system interface:

```
(File-Type-Use-Language-View-Style "col" "colander")
```

This specifies that any file whose name ends with the extension "`.col`," will be visited by *Pan* in the context of the view style named "`colander`." *Pan*'s default configuration file contains similar declarations for standard file extensions, but they may be overridden and extended at any time. Should *Pan* be extended (as its design permits) with an interface to another kind of persistent store, a similar declaration would map types (in whatever sense supported by the store) to view styles.

Personal customization code may alter a standard language-based view style configuration by changing the values of options and other bindings. For example, in a file named `mlv-colander-mode.cl` the following form would request that a customization file be loaded immediately after the loading of view style "colander" (or at editor startup time if the view style is preloaded).

```
(Add-Language-View-Style-Auto-Require "colander" "mlv-colander-module")
```

The customization file mlv-colander-module.cl might then override configurations with the following form. The characters "<>{}*" are metacharacters in this and all other examples in this section.

```
(With-View-Style-Scope (view-style colander-view-style)
  ...<set options> )
```

B.2 The View Style Specification File

A language-based view style specification resides in a file named "<view style>-lang" located somewhere on the system's search path, where "<view style>" is the view style's name. In the provisional specification mechanism, the specification file is loaded directly by COMMON LISP, so it may contain mix of traditional *Pan* extension code and declarative specifications in S-expression form. The aim, of course, is to minimize the amount of auxiliary procedural code that is needed for routine cases, but to retain flexibility to prototype new variations that cannot be expressed (yet) by the specification language. Eventually, auxiliary COMMON LISP code will reside in separate COMMON LISP source files, just as they do now for *Colander* descriptions.

B.3 The Define-Language-View-Style Form

The declarative part of a view style configuration file consists of S-expressions in the provisional specification language. This syntax amounts to the intermediate form of a specification language that has not yet been implemented. These expressions are wrapped in the top-level form Define-Language-View-Style, a COMMON LISP macro that processes the specifications. Building the intermediate form first permits early development of the back-end mechanisms necessary to process the specifications. This intermediate form is analogous to *Colander*'s PLI intermediate form, as produced by the *Colander* preprocessor.

In this intermediate language, a complete and workable *Pan* language-based view style is defined by an instance of following form:

```
(Define-Language-View-Style <name>
  {<specifier>}* )
```

The string "<name>" identifies the language; the detailed nature of a "<specifier>" will be described below. A complete example for Modula-2 appears in Appendix A. Loading a file containing this top level form has several effects:

1. it loads *Ladle* and *Colander* descriptions, using conventions for names and locations of files;

2. it creates a new view style (a configuration binding scope); and

3. it configures the newly created view style according to the list of specifiers.

As always with *Pan* scoping, considerable default behavior may be inherited from the global scope.

B.4 View Styles, Object Classes, and Scopes

The top-level form `Define-Language-View-Style` automatically creates a view style named "`<name>-view-style`," where "`<name>`" is the view style name supplied to the form. The existence and name of this view style may be relevant to end users and designers (see above how one might add personal language-specific customizations). Nothing else in this section needs to be exposed to end users or designers.

Loading the `Define-Language-View-Style` form also also creates new language-specific subclasses (in the CLOS sense) of edit objects and views. These are part of the implementation, but they generally need not be exposed to designers. These are:

- a new edit-object class named "`$<name>-edit-object$`"
- a new view class named "`$<name>-text-view$`"

where "`<name>`" is the view style name supplied to the form. These classes are subclasses of `$lb-edit-object$` and `$lb-text-view$` respectively, both of which carry methods for generic language-based functionality (even though there will never be instances of the two superclasses).

Figure 4.4 on page 45 shows *Pan*'s edit object class hierarchy, and Figure 4.5 on page 47 the hierarchy for view classes.

Loading the `Define-Language-View-Style` form additionally:

1. establishes an internal map between the file type (e.g. "`col`") and the edit-object class for visited files of that type;
2. makes `<name>-view-style` the default view style for all visited files of that type.

B.5 View Style Specifiers

A "specifier" in the intermediate specification language is a COMMON LISP form (list) that begins with an identifying keyword and whose subsequent arguments are suitable to the particular specifier. Arguments may be symbols (unquoted names), COMMON LISP data literals, lists of arguments, and keywords. For example, the specification fragment in Figure B.1 establishes simple key bindings for the view style; the fragment in Figure B.2

```
(:bind-keys
 Set-Oplevel-To-Language-Error          "^C !"
 Set-Oplevel-To-Query                   "^C q")
```

Figure B.1: Key Binding Specification

sets option values in the view style; the fragment in Figure B.3 defines an operand class (and level, partially) for the "Colander" view style; and the fragment in Figure B.4 specifies the standard menu for the "Colander" view style.

Specifiers currently defined appear in Table B.1; see Appendix C for documentation on each.

```
  (:option-values
   Text-Window-Tabwidth                   7
   Announce-Error-Counts-After-Analysis   nil)
```

Figure B.2: Option Value Specification

```
(:define-operand-classes
 (colander-decl-opclass
    :title         "Declaration"
    :documentation "Colander declarations."
    :definition    (:operators "fact_decl"
                               "entity_property_decl"
                               "maintained_property_decl"
                               "node_property_decl"
                               "entity_name_decl"
                               "datapool_name_decl"
                               "pure_decl"
                               "mode_decl")
    :bind-structural-commands
    ;; Use a standard operand menu titled "Declaration"
    :operand-menu       user::declaration-oplevel-menu
    :before             lang:Analyze-Quietly))
```

Figure B.3: Opclass Specification for "Colander Declarations"

```
(:default-menu
 (:copy-menu user::default-language-menu "Colander"
             ((:copy-menu user::global-language-menu "Language"
                          ((:level-menu "Levels"))
                          ((:highlighter-menu "Highlighters")))
              nil 4)
             ((:menu "Preprocess"
                     (Save-Write-PLI-File "Save/Write-PLI File")
                     (Write-PLI-File "Write-PLI File"))
              nil 5)))
```

Figure B.4: Menu Specification for "Colander"

```
(:add-char-matches {<left-char> <right-char>}* )
(:add-char-sets {<char-set-name> <char-list>}* )
(:add-defaults)
(:add-flags {<flag-descriptor>}* )
(:after-analysis-function <function> )
(:before-analysis-function <function> )
(:bind-keys {<command> <key-descriptor>}* )
(:default-menu <menu-spec> )
(:define-highlighters {<highlighter-description>}* )
(:define-operand-classes {<class-description>}* )
(:delete-char-matches {<left-char> <right-char>}* )
(:delete-char-sets {<char-set-name> <char-list>}* )
(:delete-flags {<flag-descriptor>}* )
(:highlighters {(<highlighter-name> <:on|:off> <key-spec>* )}* )
(:init-function <function> )
(:operand-levels {(<operand-class-name> <key-spec>* )}* )
(:option-values {<option-name> <value>}* )
(:sde-oplevel-menus <oplevel-menu-name> {<opl-menu-spec>}* )
(:shadow-char-matches {<left-char> <right-char>}* )
(:shadow-char-sets {<char-set-name> <char-list>}* )
(:shadow-flags {<flag-descriptor>}* )
```

Table B.1: Language-Based View Style Specifiers

```
(setf (variable ...)  ..)
Add-Default-Language-Configuration-To-View-Style
Bind-Key
Bind-Menu
Bind-Oplevel-Command
Bind-Oplevel-Menu
Copy-Menu
Create-Menu
Define-Highlighter
Define-Operand-Class
Define-Structural-Oplevel-Commands
Set-Char-Match
Set-Char-Set
Window-Modify-Flag-Collection
```
and some view methods

Table B.2: Configuration Forms That Implement Specifiers

B.6 Specifier Implementation

Specifiers are implemented by macro expansion into appropriate calls on existing *Pan* configuration mechanisms, including those listed in Table B.2.[1]

The implementation of `Define-Language-View-Style` ensures that all of these are evaluated in the proper context: in the scope of the new view style so that bindings will be added correctly, and with the *Ladle* global syntactic state to the newly loaded language so that operator names in the *Ladle* description may be named and processed correctly.

B.7 Creating New Specifiers

The implementation of `Define-Language-View-Style` defines no specifiers directly; all are created dynamically via calls to a defining form, so more can be added at any time. This feature is of special value during development of the intermediate language and its supporting mechanisms, but it might also be of use by designers and users.

For example, the experimental module for syntax directed editing (module `sde.cl`) is normally loaded from the library only when needed (possibly at runtime), typically along with the first language-based view style that uses it. When loaded, the `sde` module extends the view style specification language by adding the new specifier `:sde-oplevel-menus`. A client view style need only ensure (via a Common Lisp `require`) that the `sde` module be loaded before attempting using this specifier to create menus that support placeholder expansion templates.

It is not clear whether this degree of flexibility can persist usefully after the addition of a front-end syntax with preprocessing.

[1] In many cases the internal equivalent of the configuration form is used.

For example, an unusually simple definition (one with no argument checking at all) is:[2]

```
(def-language-description-specifier :add-flags
  "(:add-flags {<flag-descriptor>}* )
Add all specified flags to the flag collection in the view
style of the language. See Window-Modify-Flag-Collection."
  #'(lambda (arglist)
      `((win:Window-Modify-Flag-Collection :view-style
                                           :add
                                           ',arglist))))
```

This translates a specification of the form

```
(:add-flags <flag specs>)
```

into

```
(win:Window-Modify-Flag-Collection :view-style
                                   :add
                                   <flag specs>)
```

which is how an equivalent specification for some other view style might appear in the standard configuration file `default.cl`.

A slightly more complex definition is:

```
(def-language-description-specifier :bind-keys
  "(:bind-keys {<command> <key-descriptor>}* )
For each pair in list, bind <command> to the key sequence
<key-descriptor> in the view style of the language.
See Bind-Key."
  #'(lambda (arglist)
      (do ((in-forms arglist (cddr in-forms))
           (out-forms (create-empty-queue)))
          ;; loop termination
          ((or (null in-forms)
               (length=1 in-forms))
           (when (length=1 in-forms)
             (Editor-Warn ":bind-keys ignoring odd argument ~S"
                          in-forms))
           (list-queue-elements out-forms))
          ;; loop body
          (let ((cmd (first in-forms))
                (keyspec (second in-forms)))
            (enqueue `(Bind-Key ',cmd ,keyspec :view-style)
                     out-forms)))))
```

Some specifiers are rather more complex, especially those that accept complex nested specifications. Each implements what amounts to a compiler, along with error diagnosis and recovery, for a "little language."

[2]Note that this definitional form takes a string argument containing documentation; this is incorporated by *Pan*'s online help system.

Appendix C

Language-Based View Style Specifiers

This appendix contains documentation that was generated automatically from information collected by def-language-description-specifier. Similar information is available in *Pan*'s help view (see the "Documentation" submenu).

In many cases, the documentation here refers the reader to documentation for other definitional forms in the language. These are available in *Pan*'s online documentation, and the services are described in more detail the *Pan* user manual [31]. The characters "<>{}*" are metacharacters in this documentation.

(:add-char-matches {<left-char> <right-char>}*)

For each pair of characters, add their definition as a matching pair in the view style of the language. See Set-Char-Match for more documentation about character matching.

(:add-char-sets {<char-set-name> <char-list>}*)

For each pair, add every character in <char-list> to the character set <char-set-name> in the view style of the language. See Set-Char-Set for more documentation about character sets.

(:add-defaults)

Adds the default bindings for general language based editing to the view-style of the language. These may be overridden piecewise by subsequent specifications.

(:add-flags {<flag-descriptor>}*)

Add all specified flags to the flag collection in the view style of the language. See Window-Modify-Flag-Collection for more documentation about panel flags.

(:after-analysis-function <function>)

Add a view method function to be run after each analysis; it is called in the full context of <view>. Note that the definition of this function will not be evaluated with the new language syntax current, so it should contain no load-time grammar transformations. See defgeneric of view-notify-after-analysis for more documentation on this generic function.

(:before-analysis-function <function>)

Add a view method function (of one argument <view>) to be run before each analysis; it is called in the full context of <view>. Note that the definition of this function will be not be evaluated with the new language syntax current, so it should contain no load-time grammar transformations. See defgeneric of view-notify-before-analysis for more documentation on this generic function.

(:bind-keys {<command> <key-descriptor>}*)

For each pair in list, bind <command> to the key sequence <key-descriptor> in the view style of the language. See Bind-Key for more documentation about key bindings.

(:default-menu <menu-spec>)

Use the menu described by <menu-spec> as the default menu in the view style of the language. <menu-spec> may take several forms:

- <menu-name>

  A symbol specification causes the named menu, presumed to exist already, to be used.

- (:menu <title> { (<entry> [<label>]) }*)

  A newly created menu will be used, to appear with <title>, a string, at the top. Any number of bindings may be follow, each of which will be added in order to the newly created menu. <entry> may be one of:

  - the name of a bindable editor command,
  - the name of another menu to be added as a pullright submenu,
  - a <menu-spec> recursively, or
  - the keyword :non-selectable.

  The entry will be labeled with the string <label>, if given, or a default. See Bind-Menu for more documentation on new menu creation.

- (:copy-menu <menu-name><title> { (<entry> [<label> [<position>]]) }*)

  A copy will be made of the menu, presumed to exist already, named by the symbol <menu-name>, and the copy will appear with <title>, a string, at the top. Additions will be made to the new menu as specified by any number of additional bindings. Each binding is specified by <entry>, as described above. The entry will be labeled with the string <label>, if given, or a default if <label> is missing or nil. The entry will be inserted at <position>, an integer, or at the end of the menu if missing. See Bind-Menu for more documentation on menu copying.

- (:level-menu [<title>])

  A menu will be constructed, to appear with <title>, a string at top; default title is "Set Levels". Bindings will automatically be added for each of the current operand levels for which a setting command has been defined. Each such binding will appear as an entry whose title is the title of the operand level, annotated with a key binding, if any, that invokes it in the local context.

- `(:highlighter-menu [<title>])`

  A menu will be constructed, to appear with `<title>`, a string at top; default title is "Toggle Highlighters". Bindings will automatically be added for each of the currently available highlighters for which a toggling command has been defined. Each such binding will appear as an entry whose title is the title of the highlighter, annotated with a key binding, if any, that invokes it in the local context.

`(:define-highlighters {<highlighter-description>}* )`

Define a new highlighter for each `<highlighter-description>`. A highlighter associates with an operand class a visual effect that, when a highlighter is activated, will be maintained automatically. A `<highlighter-description>` has the form:

```
(<name>
 :opclass <operand-class-name>
 [:title <title>]
 [:documentation> <documentation>]
 [:apropos <apropos>]
 [:effect :fg1|:fg2|:fg3|:bg1|:bg2 ]
 [:toggle-command <toggle-command-name>]
 [:strict? t|nil ]
 )
```

- `<Operand-class-name>` is a symbol naming a previously defined operand class. This class defines which nodes will be highlighted.

- `<Title>` is a string by which the highlighter is described to users, for example, in a menu. The title need not be unique; for example many languages may have a level titled "`Statement`". Default is the name of the operand class.

- `<Documentation>` is a descriptive string.

- `<Apropos>` is a symbol or a list of symbols. Apropos is generated for the specified symbols as well as for the name of the highlighter.

- `<Effect>` must be one of the specified keywords. The three foreground effects use ink colors 1 through 3 respectively, as configured by option Text-Window-Fg-Colormap. The two background effects use shading colors 1 and 2, as configured by option `Text-Window-Bg-Colormap`. Default is `:fg1`.

- `<Toggle-command>` specifies the name of the bindable command that toggles the highlighter in current view between active and inactive. The default, when there is no toggle-command specification, is to create automatically a command named `Toggle-<name>`. That name may be overridden by this specification, or, if `<toggle-command>` is `nil`, the creation of the command may be suppressed entirely.

- `<Strict?>` specifies whether the effect will be cleared when text and analyzed data become inconsistent. Default is `t`.

See Define-Highlighter for more documentation on highlighters.

(:define-operand-classes {<class-description>}*)

Define a new operand class for each class specification <class-description>. An operand class describes a dynamically determined subset of the tree nodes in a document, and associates certain kinds of useful behaviors with operations on those nodes. A <class-description> has the form:

```
(<name>
 :definition <test-spec>
 [:title <title>]
 [:documentation> <documentation>]
 [:apropos <apropos>]
 [:extension <extension> ]
 [:set-command <set-command-name>]
 [:bind-structural-commands]
 [:operand-menu <menu-spec>]
 [:operand-menu-command <menu-command>]
 [:before <before-daemon>]
 [:after <after-daemon>]
 )
```

- <test-spec> describes the meaning of the operand class by defining a test (i.e. a predicate) for deciding dynamically whether a given node is a member of the class or not.<test-spec> is a possibly nested list in one of the following forms:
  - (:operators {<operator-name>}*)

    The arguments form a standard *Ladle* phylum specification, where each <operator-name> is a string that names an operator in the language. Only nodes whose operator is listed are members.
  - (:parent-operators {<operator-name>}*)

    The arguments form a standard *Ladle* phylum specification, where each <operator-name> is a string that names an operator in the language. Only nodes whose parent operator is listed are members.
  - (:grandparent-operators {<operator-name>}*)

    The arguments form a standard *Ladle* phylum specification, where each <operator-name> is a string that names an operator in the language. Only nodes whose grandparent operator is listed are members.
  - (:predicate <predicate>)

    <predicate> is a function (or the name of a function) taking one argument: a tree node. The function is guaranteed to be run with correct global language state, both syntactic and semantic.
  - (:member-variable <variable-name> [<test>])

<variable-name> is the symbol name of a *Pan* variable, as defined by one of Define-Variable, Define-Option-Variable, or Defvariable. Class membership is computed dynamically by evaluating a particular tree node for membership in the list that is the current value of the variable. <test> is a function of two arguments used to determine membership, default is #'eq.

  - (:or {<test-spec>}*)
    (:and {<test-spec>}*)
    (:not <test-spec>)

    <test-spec> may be a nested specification. Membership is determined by combining the included specifications into a compound test, using standard Lisp logical operations.

- <Title> is a string by which the level is made visible to users, for example, in a menu. The title need not be unique; for example many languages may have a level titled "Statement."

- <Documentation> is a descriptive string.

- <Apropos> is a symbol or a list of symbols. Apropos is generated for the specified symbols as well as for the name of the level.

- <Extension> is a symbol or a list of symbols, each of which names a *Pan* scoped variable, created using Define-Variable, Define-Option-Variable, or Defvariable. This provides an alternate definition of class membership that can makes some internal operations much faster. A single variable name identifies a variable whose value can always be relied upon to be a list of nodes that enumerate the membership of the class; when more than one variable is named, the set union of their values defines class membership. The extensional definition is optional, and when present, no attempt is made to verify equivalence between it and the predicate (<text-spec>) definition.

- <Set-command> specifies the name of the bindable command that sets the current view's oplevel to this oplevel (a panel menu permits them to be set too). The default, when there is no set-command specification, is to create automatically a command is named Set-Oplevel-To-<name>. That name may be overridden by this specification, or, if <set-command> is nil, the creation of the command may be suppressed entirely.

- The presence of the :bind-structural-commands keyword specifies that operand bindings for this level be created for the standard set of structural commands, see Define-Structural-Oplevel-Commands for more documentation.

- <Menu-spec> describes a menu to bind as the designated operand level menu for this operand class. The specification may be a symbol naming a preexisting menu, or it may be a recursive specification of a new menu, as with the argument to :default-menu. See documentation for specifier :default-menu (above) and for Bind-Oplevel-Menu for more documentation on operand level menu bindings.

- <Menu-command> replaces the normal operand menu dispatch command; this causes <menu-name> to be ignored unless handled specially by the replacement.

- `<Before-daemon>` is a function of no arguments. It will be called before each execution of an operation dispatched at the new level.
- `<After-daemon>` is a function of no arguments. It will be called after each execution of an operation dispatched at the new level.

See `Define-Operand-Class` for more documentation on operand class definition.

```
(:delete-char-matches {<left-char> <right-char>}* )
```

For each pair of characters, delete their definition as a matching pair in the view style of the language. See `Set-Char-Match` for more documentation about character matching.

```
(:delete-char-sets {<char-set-name> <char-list>}* )
```

For each pair, delete every character in `<char-list>` from the character set `<char-set-name>` in the view style of the language. See `Set-Char-Set` for more documentation about character sets.

```
(:delete-flags {<flag-descriptor>}* )
```

Delete all specified flags from the flag collection in the view style of the language. See `Window-Modify-Flag-Collection` for more documentation about panel flags.

```
(:highlighters {(<highlighter-name> <:on|:off> <key-spec>* )}* )
```

Specify the names of highlighters to be made available to the user. Each instance of `<operand-class-name>`, a symbol, names a level, and these choices appear in a panel menu. When one more instances of `<key-spec>`, a string containing a keystroke specification, are included, the command that toggles the highlighter is bound to those keystrokes.

```
(:init-function <function> )
```

Add a view method function (of one argument `<view>`) to be run in the scoping context, not the full context, of each newly created primary view for the language, after all other initialization is done. Note that the definition of this function will be not be evaluated with the new language syntax current, so it should contain no load-time grammar transformations. See defgeneric of `view-initialize` for more documentation on this generic function.

```
(:operand-levels {(<operand-class-name> <key-spec>* )}* )
```

Specify the names of operand classes to be made available to the user as possible "operand levels," which control some kinds of language-based interaction. Each instance of `<operand-class-name>`, a symbol, names a level, and these choices appear in a panel menu. When one more instances of `<key-spec>`, a string containing a keystroke specification, are included, the command that sets the current level to this class is bound to those keystrokes. Any attempt by users to set the level to an operand class not declared in this way will signal an error.

```
(:option-values {<option-name> <value>}* )
```

For each pair in list, set `<option-name>` to `<value>` in the view style of the language. See `Define-Option-Variable` for more documentation on *Pan* options.

`(:sde-oplevel-menus <oplevel-menu-name> {<opl-menu-spec>}* )`

Specify a mapping between operators in the grammar and special operand menus for syntax-directed editing with templates. Each operand menu described by an `<opl-menu-spec>` is created automatically by copying the menu named by `<oplevel-menu-name>` and adding a submenu for template expansion appropriate to the operator. Each `<opl-menu-spec>` has the form:

`(<operator-name> <template-menu-title> {<template-entry>}* )`

where `<operator-name>` is the string name of an operator in the *Ladle* grammar for the language, and `<template-menu-title>` is the string that will appear as the name of the menu containing templates. Each `<template-entry>` has the form:

`(<template-entry-label> {<template-string-component>}* )`

where `<template-entry-label>` is the string name of the menu entry for invoking the template and the `<template-string-component>` strings make up the template itself. When a template contains more than one string, during expansion a newline will be inserted before each component after the first.

`(:shadow-char-matches {<left-char> <right-char>}* )`

For each pair of characters, shadow their definition as a matching pair in the view style of the language. See `Set-Char-Match` for more documentation about character matching.

`(:shadow-char-sets {<char-set-name> <char-list>}* )`

For each pair, delete every character in `<char-list>` from the character set `<char-set-name>` in the view style of the language. See `Set-Char-Set` for more documentation about character sets.

`(:shadow-flags {<flag-descriptor>}* )`

Shadow all specified flags from the flag collection in the view style of the language. See `Window-Modify-Flag-Collection` for more documentation about panel flags.