May 1987

Report No. STAN-CS-87-1159
Also numbered CSLI-87-81

Muir: A Tool for Language Design

by

Terry A. Winograd

Department of Computer Science

Stanford University
Stanford, CA 94305

Muir: A Tool for Language Design

Terry Winograd*

Abstract

Muir is *a language* ***design environment,*** intended for use in creating and experimenting with languages such as programming languages, specification languages, grammar formalisms, and logical notations. It provides facilities for a. language designer to create a language specification, which controls the behavior of generic language manipulating tools typically found in a. language-specific environment, such as structure editors? interactive interfaces, storage management and attribute analysis. It is oriented towards use with evolving languages, providing for mixed structures (combining different languages or different versions), semi-automated updating of structures from one language version to another, and incremental language specification. A new hierarchical grammar formalism serves as the framework for language specification, with multiple presentation formalisms and a unified interactive environment based on an extended notion of edit operations. A prototype version is operating and has been tested on a. small number of languages.

1 Introduction

In the past few years it has been widely recognized that programming can be greatly facilitated by a 'programming environment' designed specifically

*Muir has been developed at the Center for the Study of Language and Information under a grant from the System Development Foundation. Contributors to the research have included Raul Duran, Bradley Hartfield, Olaf Henjum, Mary Holstege, Birgit Landgrebe, Ole Lehrmann Madsen, Kurt Nørmark, Greg Nuyens, Liam Peyton, and Terry Winograd. More details are provided in a series of CSLI informal notes, and in particular in the dissertation by Nørmark [1987].

for a given language. Such environments provide specialized tools for creating, modifying and presenting programs, and for analyzing, compiling and running them.

The earliest examples, such as the Interlisp environment [Teitelman, 1969] and the Cornell Program Synthesizer [Teitelbaum and Reps, 1980], were built for use with a particular language. This continues to be the case for many commercial systems, in which considerations of the target language have guided design all the way down to the hardware level. There has also been interest in providing a more general framework for automated development of language-specific environments. Systems such as Mentor [Donzeau-Gouge et al., 1980, 1984], Gandalf [Notkin, 1985], and POPART [Wile, 1982] allow an environment-designer to specify the language in a principled way, so that the specification can be used to semi-automatically generate the environment. These more general tools are applicable to more than programming languages. Any suitably structured language can be the basis for an environment. In particular, each of these systems has been used to create an environment for working with its own language-describing language ('meta-language').

Our own work takes place in Stanford's Center for the Study of Language and Information. Work at that center includes research on programming languages and specification languages; the development of formal languages (and correlated theories) for representing meanings of natural language; the development of grammar formalisms (in conjunction with syntactic theories of natural language); and frameworks for providing language-composition tools, such as text editors and formatters. Much of the work is done using a common software base provided by Xerox 1100-series workstations, running Interlisp-D.

This wide range of enterprises offers an opportunity to develop tools that apply general principles to the common problems of language creation and use that come up in diverse areas. The research described here was motivated by work on one particular language (a system specification language called Aleph [Winograd, forthcoming]) but has a long-term goal of being applied to many of the different kinds of languages relevant to CSLI and more generally to the broad cross-section of research it represents.

In our situation? the primary users are actively engaged in the design and development of formalisms (languages). This is not the ordinary situation in which a designer builds an environment for a fixed language (e.g., Pascal) which is then used by many people. Our use of language-based tools (editors, file systems, etc.) includes experiments which lead to continuing revisions and redesigns of the language. It is therefore important to support the process of language design and modification, slanting the implementation of the environment to make this as productive as possible, rather than focussing on the efficiency provided to ultimate end users of a stable language. We call such an environment a. ***language development environment*** (LDE).

In a LDE, language designers can make incremental changes to a language (both its surface syntax and underlying structure and semantics) and have the environment change accordingly. For example, the structure editor will always produce structures corresponding to the currently defined grammar, and tools are available to translate older structures into the corresponding updated versions. At any moment in the development process, the environment must provide a reasonable base for working with the language in order to test it. That is, it must provide a realistic language-user environment within the larger context of language development. For widely used languages, we would expect the LDE-based environment to be replaced with a production-quality language-specific environment once the language was stabilized. The price paid for the generality and incrementality of a LDE might not be appropriate for someone simply concerned with using the language in a fixed state, although it might often be a better alternative than exerting the effort needed to make a specialized efficient environment. We could imagine tools for 'compiling' the language specification into a language-tuned environment, but we are not currently dealing with this issue. We are more concerned with providing a tool that makes sense for language designers and developers.

We have assumed that the users of a LDE will be relatively experienced in its use, but not necessarily experienced in LISP programming. The goal is to provide a framework and a set of linguistically-oriented formalisms which enable the language designer to specify a language at an appropriately high level, with escapes into implementation code only in special cases. We are also assuming that the environment will be a tool, not an intelligent agent. We are not concerned with trying to develop 'intelligent' algorithms, but with automating the frequently occurring routine tasks, leaving more specialized and intellectual tasks for the person in interaction with the system.

Our choice of implementation vehicles was determined largely by the pool of existing equipment and interest, and has both its benefits and its costs. The benefits lie in the rich set of resources available in Interlisp-D, both in the system itself and in a wide selection of user and library 'packages'. It has been possible to develop a system that makes full use of an interactive high-resolution graphic workstation, with windows, menus, multiple fonts and type sizes, graph and text editors, remote file servers, etc. without having to do the underlying systems work ourselves. This has made it possible to reach a significant level of performance with a relatively small amount of programming. The disadvantages include the difficulties of achieving adequate performance on the smaller 1100-series machines, and the lack of transportability. Something done in C on UNIX™ clearly has much wider use. This limitation is not critical given our overall research strategy-that is, to explore the possibilities for a LDE in a highly fluid research prototype system. We expect this work to be the basis for later, more widely accessible systems.

We will first summarize the basic design problems facing Muir and describe the design directions that we are pursuing. A more detailed account appears in the following sections.

1.1 Problems

1. **Create a basic language definition framework that is powerful enough to handle a broad range of formal languages.** We want to be able to include existing and evolving programming and specification languages, logical notations (such as predicate calculus and situation semantics), and linguistic formalisms. A user of the system should be able to write a.language specification that is principled and high level, and which can be used in table-driven way to provide a wide range of facilities.

2. **Provide a collection of generic tools that take advantage of the interactive LISP environment.** Working from a language specification, the system should provide interactive means of creating, viewing (in multiple ways), storing, modifying and analyzing structures in the language. To the greatest degree possible it should be incremental (allowing local changes to take effect without a high-overhead compilation cycle) and WYSIWYG' (operations on structures appearing on the screen are directly linked to changes to underlying structures, without needing to think about intermediate translation).

3. **Provide for changes in the languages being specified.** This means, for example, that a structure built using one version of a language specification should be easily modified to satisfy a. later version. As much as possible, changes should be isolated so that they do not require major restructuring or analysis

4. **Make it possible for structures in the target languages to be integrated with other texts.** It should be possible to use the Muir environment to produce a structure (e.g., a program in a programming language) and then to include it in a paper being written using some text formatter. Conversely, it should be possible to take a text from some other context and to integrate it into the structures.

5. **Provide for the sharing of language structures** (including language specifications) among several researchers, dealing with issues of access control and version control.

1.2 Basic Design Decisions

1. **Uniform abstract syntactic structure model based on a hierarchical grammar with properties and attributes.** The system is based on the manipulation of abstract syntax trees. The 'real source' of a structure in a language is not a text but a tree-organized data structure. This is like Mentor and Gandalf, and unlike traditional environments that are text-based, or environments such as OMEGA [Linton, 1984] and POPART [Wile, 1982], which build on an object-oriented or data-base representation. The formalism allows for mixed structures, in which a tree can include elements from more than one language.

2. **Separation of abstract structure and presentation.** The underlying structure for objects (texts, documents, programs, etc.) in a language does not include any information about how it will appear (including key words, punctuation, ordering etc.). These are specified separately and there can be more than one for a given abstract grammar. Presentations include formatted text and graph-structure diagrams.

3. **Interaction via generic structure editors that can be used to present and modify structures on the basis of a variety of visual presentations** (formatted text, graph diagrams, etc.). These editors use a uniform structure in which a variety of edit operations are available for each type of structure, providing both environmental operations (e.g., opening new display windows) and structure modifications. Some of these operations call on other generic tools for transformation, at tribute evaluation and storage management. Along

with the separation of structure from presentation, this makes it possible to unify activities treated separately in many environments, such as editing and browsing.

2 The Language Specification

First we will give a summary paragraph, introducing terms that will be defined below.

In Muir, a person manipulates ***structures*** that are in a ***formalism*** defined by a ***language specification*** (LS). These structures are organized around ***abstract syntax trees*** (AST) which conform to a ***core grammar*** which is a part of the LS and to ***imported Ianguages*** declared in the LS. Associated with nodes of these AST trees are ***properties,*** some of which are declared in ***property declarations,*** and some of whose values are derived according to ***attribute equations*** provided in the LS. Structures are mapped onto visual representations (on a screen or paper) according to ***presentation schemes*** given in the LS. A language may include any number of different presentation schemes, which are ***textual, tree,*** or ***graphic.*** A textual present ation scheme of appropriate form can be associated in the LS with ***a parsing scheme,*** which includes a ***token*** *recognizer.*

For example, given a language specification for a programming language, a user would manipulate structures (programs and program fragments) that are allowable according to the language definition. Note that we define the formalism as a set of structures, not the set of strings that can be peeled from the leaves of those structures (as in the usual definition of languages). Since everything in Muir operates with these structures, they are taken as primary.

2.1 The Core Grammar

The core grammar for a language corresponds in power to a context-free grammar. However it is based on a hierarchical formalism, which extends previous phylum-operator models [Donzeau-Gouge et al., 1980; Reps and Teitelbaum, 1984; Medina-Mora, 1982] through a mechanism of inheritance as found in object-oriented languages with subclasses, such as Simula and Smalltalk. Non-terminal symbols of a CFG correspond to phyla, which in turn have associated operators. The formalism is defined in the appendix. We will describe less formally several basic aspects:

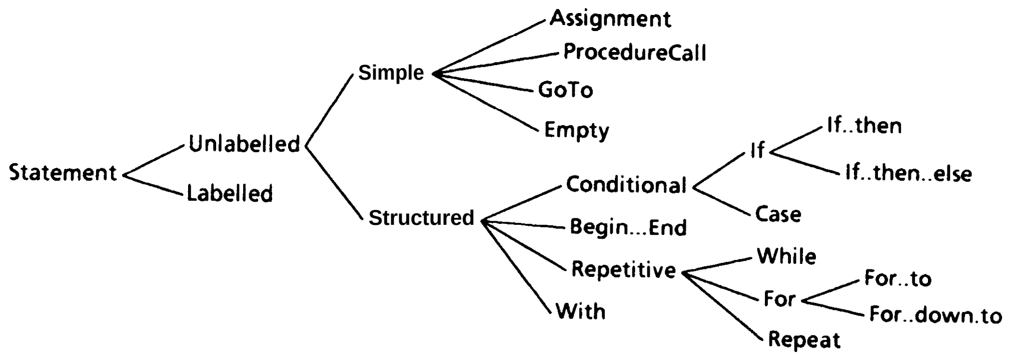

Figure 1: Partial phylum hierarchy for a Pascal Grammar.

1. **Phyla are organized into a directed acyclic graph** (a hierarchy allowing multiple parents). Each ***terminal*** *phylum* (having **no** descendants) in this graph has an associated operator, which is comparable to the right hand side of a. rule (it uses names rather than ordering to distinguish the constituents). The non-terminal elements of the graph are called *categorial phyla.* A phylum A is a *subphylum* of B if there is a. path through the directed graph from A to B (including the zero-length path). Language definition information (attribute equations, edit operations, etc.) can be associated with categorial phyla and are inherited by all of their subphyla.. There is a. l-l mapping of terminal phyla to operators. Figure 1 illustrates part of the phylum hierarchy for the Pascal language.

There is a simple mapping from this onto a ***weakly equivalent*** CFG: Each terminal phylum corresponds to a rule whose left hand side is the the phylum name, and whose right hand side is its operator. Each categorial phylum corresponds to a set of rules each having as its left-hand-side the categorial phylum name, with a right hand side consisting of a. single symbol naming one of its immediate subphyla.

The motivation for this mechanism lies in our interest in the structures, rather than the generated strings. The phylum/operator structure is not strongly equivalent to the corresponding CFG structure, as illustrated in Figure **2.**

The CFG version includes layers of single-constituent phrases, in order to reflect the subclass structure. Of course someone using a standard BNF would be unlikely to include such a fully articulated hierarchy. We make constant use of it as the conceptual organization for associating operations and other information with grammatical categories. The phylum/operator formalism makes this possible without introducing superfluous nodes in the trees. There are also special facilities for dealing with list structures (e.g., for displaying and editing them) even though they are represented in the underlying trees in standard binary list fashion.

2. **Phyla can be imported from other languages.** An operator can include symbols corresponding to phyla in its own language, and also ***gateways***, which specify an imported language and a phylum in that language. Categorial phyla can include imported phyla among their alternatives. This makes it possible to have ***mixed structures*** in which one part of the structure is in a separately defined formalism. This is used in several ways. First, it allows modularity in designing languages. For example, the language-specification-language (metalanguage) used in Muir includes parts for specifying the core grammar, for specifying attribute equations, for specifying edit operations, presentation rules, etc. Each of these is defined as a separate language, all of which are declared as imported languages in the metalanguage. The benefits are the same as in any modularization of what would otherwise be a large flat heterogeneous structure. Also, mixed structures are a key element in using the system for transformations from one language to another and from one language version to an updated version, as described in [Nørmark, 1986].

3. **An AST can include unexpanded nodes** which specify a phylum and have no further structure. In structure-directed editing, this is an obvious necessity, as we need to work with structures in progress. In Muir, this is supported at all levels. That is, ASTs with unexpanded nodes can be edited, stored, printed, etc. equally with fully expanded trees. Of course language-specific tools (e.g., compilers) would have

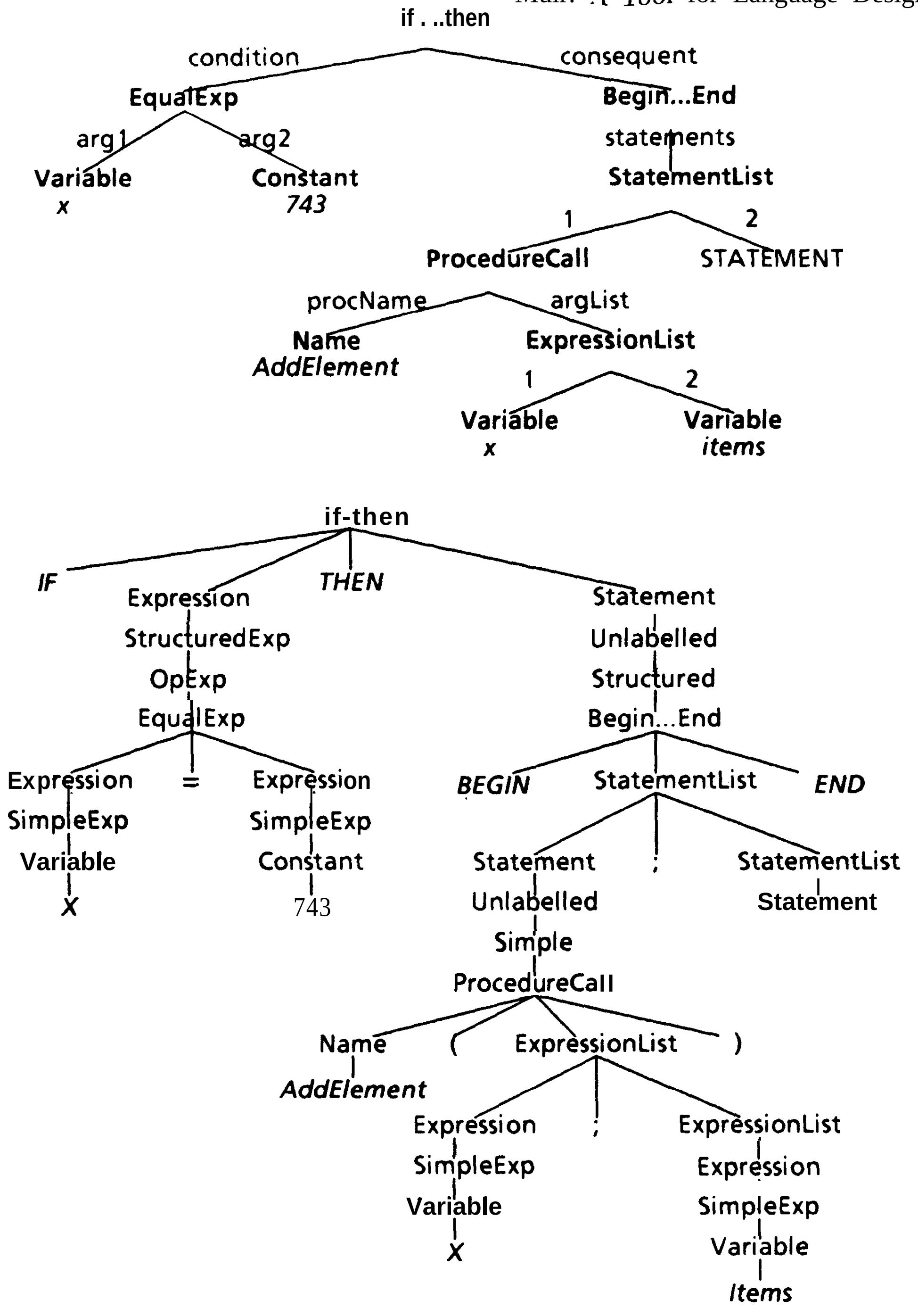

Figure ‘2: Structures produced by a phylum-operator grammar and the weakly equivalent CFG to reject them or provide for special interactions to gather the missing information.

4. **The structure as embodied in an AST is not a homomorphic image of some text.** An AST specifies the types of nodes and the set of children of each node, without giving any textual information, including order. The operator is a set of pairs, each consisting of a unique (with respect to that operator) ***constituent name*** (or 'tag'), and a phylum name. An AST node based on this operator must have exactly one child for each pair, and the child must be a node whose phylum is a terminal phylum that is a subphylum of the second element in the pair, or an unexpanded node corresponding to that element. We call the phylum specified in the operator the ***choice phylum*** and the terminal phylum used in the expansion of the child node the ***identification phylum.***

This last property goes along with a radical separation throughout the system between ***structure*** and ***present&ion.*** The core grammar defines the set of structures, but does not describe any mapping of these onto sequences of characters, graphical representations, or the like. There is a separate part of the language specification (the presentation schemes and rules) which do this, and they can do it in a variety of ways, as discussed below.

2.2 Properties and Attributes

AST structures are used as the fundamental organization of data for all purposes related to structures in a formalism. This includes data that is being used for temporary purposes (e.g., mappings between tree structures and corresponding display elements), data that has been reorganized for efficiency reasons (e.g., collecting all uses of a given variable into a list) data used by particular tools (e.g., a compiler or program analyzer), and permanently associated data that does not fit naturally into the language structure (e.g., comments). In order to do this, Muir allows property-value pairs to be associated with any AST node. These are similar to the 'annotations' used in MENTOR.

Property declarations can be associated with phyla in a language specification. The hierarchy makes it possible to associate them at different levels. For example, a property relevant to all phyla (such as 'Comment') can be declared at the root node of the phylum hierarchy (called 'Any'). There are some important differences between properties and AST constituents:

1. **Properties can have values not in the language.** Every child of an AST node must be a proper AST node in the language (as extended by the imported languages). A property value may be an AST, or may be an arbitrary data structure in the implementation language (Interlisp-D).

2. **No checking is done of value types for properties.** The structure manipulating facilities make it impossible to put a wrongly typed constituent into an AST. For properties, no checking is done. Declarations are for documentation and to determine what editor should be used when editing a property (the AST editor, the LISP structure editor or other specialized data-structure editor (eg., a bitmap editor), or a text editor).

3. **Undeclared properties are allowed.** The facilities for setting and changing property values allow undeclared property names, operating on the values according to whatever type information can be inferred from them directly.

4. **The mechanisms for presenting properties on the screen and editing them are distinct from those for constituents** (typically, but not always, involving opening up a new separate window for the property).

Property declarations are used to determine how property values should be treated in presentation, and when structures are copied (either in core or to secondary storage). A property can be ***ephemeral*** (never copy it), ***temporary*** (copy it in core, but don't write it to secondary storage), ***derived*** (with associated rules for rederivation) or ***permanent*** (always copy it). Undeclared properties are treated as temporary. Permanent properties are further declared according to whether the values are written into the same storage as the underlying tree, or into a separate module (so the tree could be reloaded without them, or with a different one).

Many analyses of a structure are best organized according to phyla (e.g, type checking, variable usage analysis, conversion to canonical forms, compilations of various kinds, etc.). That is, we can associate with each phylum a procedure (or equation) that determines the appropriate value based on values for adjacent (parent and child) nodes.' This kind of modularity is especially important in a LDE, where frequent changes to the grammar are being made. In order to reflect the changes to the corresponding analysis one generally needs only to change the procedures or equations corresponding to the modified phylum and to those operators that call it. ***Attribute grammars*** [Knuth, 1968] can be applied to organize this kind of node-local computation.

A Muir language specification can include ***attribute sets,*** each declaring a set of property names and a collection of equations used in calculating values for those properties. These are used as in ordinary attribute grammars, with additions for our extended grammar form. In particular, an equation can be associated with a categorial phylum. A terminal phylum without an equation for one of its attributes will inherit the nearest one up the hierarchy (as is usual in inheritance systems). It is an error to have equations inherited from two distinct ancestors. This and other errors in the equation set (e.g., circularities) can be checked for with an attribute-equation-checking tool [Landgrebe, 1987].

2.3 Presentation Schemes and Rules

One of the major consequences of a structure-based (rather than text based) environment is that the mapping from structures to visible representations can be 'decoupled.' This was recognized in the initial work on syntax-directored editors [Hansen, 1971], but can be much more usefully exploited in a high-resolution graphics envrionments where presentations with different graphical structures and formatting can be appropriately used.

A core grammar includes no information on the 'language' as a set of strings or graphical entities. These are given as ***presentation schemes (as***sociated with the language as a whole) and ***presentation rules*** (associated with individual phyla and inherited through the hierarchical structure). In developing a language, this makes it possible to experiment with changes to the surface syntax without changing the underlying trees and the analyzers and tools associated with them. In a fixed language, it makes it easy to provide multiple views, presentations specially suited to a medium (e.g., display screen vs. paper) or a use (e.g., special forms for inclusion in formatted documents) and presentations with associated interactive properties (e.g., graph displays in which items can be 'buttoned' to produce actions). It allows us to unify aspects of an environment normally treated separately, such as editing and browsing.

Muir currently supports two basic presentation forms, ***formatted test*** and ***tree diagrams*** (we have also begun developing more general graphical presentations). Formatted text is based on sequences of characters with associated ***looks,*** such as font, size, face, and vertical offset. These are laid out in a structure that controls indentation, line-wrapping, and other such issues normally dealt with under the heading of 'prettyprinting'. Tree diagrams present a homomorphic image of the structure tree (with nodes potentially omitted, merged, etc.) with labels (which in turn use a textual presentation). This is especially useful for larger-scale structures, such as the overall (module-level) organization of a program or specification. However there is no built-in distinction. A language specification for any language can include presentation schemes for showing its elements as trees, as texts, or both, in any number of different ways.

A presentation scheme associates a ***presentation rule*** with each terminal phylum (allowing inheritance from categorial phyla). We have approached the design of this scheme at three levels of sophistication:

1. **The simple level.** At this level, there is a straightforward mapping from sequences of elements in an operator to sequences of tokens (in the text presentation) or nodes (in the tree diagrams). A text presentation rule specifies ordering, presence of key words, punctuation, and line breaking. This produces results equivalent to the simple formatting done in most structure editors. A tree presentation rule can control collapsing (showing descendants as though they were children) and labelling.

2. **The general level.** At this level (See Peyton, 1987) the rule-writer has much finer-grained control over the presentation. There is a 'virtual machine' and a 'presentation language' that make it possible to control looks (fonts, etc.), indentation (under several standard conventions) and conditional line-breaking in the text schemes. This will produce 'prettyprinting' of the quality done in published algorithms. It also has special facilities for holophrasting and for 'comment' properties, allowing them to appear off to the side, or to be indicated with a simple mark and expanded on demand, etc. For the tree schemes, it will make it possible to present a tree that is computed from the underlying structure but is not a homomorphic image of it.

3. **The theoretical level.** Notations in computer-based languages have been extremely impoverished relative to the languages developed over the centuries for hand-writing or hand-typesetting (as in mathematics, music, and various notations used in linguistics). We tend to think of syntax as specifying linear character sequences, since that has

been the easiest way to get things in and out of computers. With the greatly increased availability of high-resolution graphical output devices (hardcopy and screen) and 2-d input devices (mice, tablets, etc.) this is no longer the case. A language design environment should make it possible to use nonlinear notations. Formatters like TEX and SCRIBE give a. user direct access to the layout of items, but these lead to focus on the page itself, not the underlying structure. In order to integrate these with the rest of the structure-based environment (the editor, storage, language-specific tools, etc.) we need to provide for principled mapping between the underlying structures and the appearance. This should not be ***ad hoc,*** but should be based on a careful analysis and rational construction of the ways in which positioning is used to reflect structure. One student is developing a dissertation in this area.

2.4 Parsing

Because Muir is a structure-based environment, the parser plays quite a different role than it does in convent ional environments , where the conversion of structure to text and text to structure play a major role in the ongoing work. In a typical programming environment, a program is kept as a sequence of characters (e.g., in a file system) and is edited using character-based editors. When something needs to be done that depends on its structure (e.g., compiling, checking, or doing structure-dependent edit operations) it must be parsed into structure. Since these typically involve large amounts of text (as in compiling) or immediate response demands (as in doing structure-dependent edit operations), the speed of parsing is essential. There is a sophisticated grammar technology for designing languages for which parsing time is linear. The only use of 'unparsing' (presentation) is in 'prettyprinters' which start from a structure and produce a text form that is intended to be more readable (for people) than the one that was parsed to produce the structure.

In a structure-based environment, the structure itself is the 'real' source. Unparsing (using the presentation schemes) takes place whenever a structure is presented (on a screen or paper), and parsing never needs to be done at all! In a 'pure' form of a structure editor, the only thing not done through structure operations is the entering of tokens corresponding to terminal nodes in the tree (e.g., identifiers). This simply needs a tokenizing algorithm, not a parser. When writing structures to secondary stable storage, there is no need to produce human-readable forms, and an efficient direct encoding of the structure can be used.

There are several problems with this ideal of never parsing:

1. **Awkward editing.** Anyone who has tried to produce arithmetic expressions with a structure editor is painfully aware that the sequence of operations needed to produce '(3*a+x)/2' involves many more actions (keystrokes, button pushes, etc.) than typing the nine characters. Languages often include particular subparts like this, where the linear presentation form is more effective. These are typically very short-an expression longer than a few dozen characters is almost impossible to type without error, and is easier to build with structure editing.

2. **Interchange compatibility.** A system that cannot parse cannot be used effectively in conjunction with other systems that are text based. For example, it is convenient to be able to produce a language text (e.g., on a home terminal) and be able to enter it into the environment.

3. **Structure transformation.** For the great majority of typical language changes, the modifications to the surface text are at least as complex as those to the underlying tree. For many changes, only the appearance changes. But in some cases the reverse is true. As an extreme example, imagine a language change which leaves the surface form untouched, but turns some previously left-branching binary list structure into the corresponding right-branching one. All the structures need to be changed, but this would be trivially done by simply writing out an unparsed version, then reading it back in with **a** parser using the new grammar.

Therefore, we need to include a parser in Muir. However, the fact that its use is limited allows us to add generality:

1. **The parser need not be highly efficient.** When used in editing, the parser is always dealing with very short texts (tens of characters). When used for structure transformation and interchange, it is typically done in an 'off-line' mode where fast turnaround is not critical.

2. **Languages need not support efficient parsing techniques.** Because of item 1, we can develop languages with full contest-free generality. This may be useful in experimenting with language designs,

or even as a final outcome for languages whose applications are not parsing-intensive.

The information contained in the core grammar is not sufficient for parsing (it does not correspond to any particular surface form). The presentation rules provide the basic information needed to produce corresponding parsing rules, although they allow a generality that makes fully automatic inversion impossible. Therefore, a textual presentation scheme (we have not considered parsing from graphics) can be used to derive corresponding parsing rules under two conditions:

1. **The presentation rules clearly identify all the constituents.** This means that the presentation specification language is used in a particular fashion in which the sequence and identity of substructures is clearly marked. Features such as conditional presentation can be used in only limited ways (i.e., the conditions must be evaluable at parse time, without the full structure available).

2. **There are no formatting-resolved ambiguities.** In dealing purely with presentation rules, we may use formatting to convey syntactic distinctions. For example, '**1234**' might be the presentation of a number, while '*1234*' is the presentation of a character string. Or indentation might be used to indicate scoping without explicit markers, as in distinguishing the two forms:

```
If something                        If something
 then if other                       then if other
        then this                           then this
 else that                                  else that
```

   For a presentation scheme to support parsing, such differences cannot have structural significance. Note that we may have several presentation schemes for a grammar, one of which makes structural use of formatting (e.g., for hardcopy publication) while another doesn't (e.g., for use in the editor).

3. **A tokenizing scheme is provided** that allows the parser to determine the sequence of tokens corresponding to a sequence of characters.

4. **Reserved words are declared in the presentation rules.** In typical languages, certain tokens are reserved for marking the syntactic structure, and cannot be used as identifiers. For example, in PASCAL one cannot have variable named 'if' or a procedure named 'begin'.

3 The Interactive Environment

As outlined in the introduction, Muir provides an interactive environment on a personal workstation (Xerox 1100 series) with a high resolution bitmapped screen and pointing device (mouse). To a large extent, the environment derives its characteristics from the underlying Interlisp-D environment, including its ways of allocating screen space (overlapping windows), selecting commands (pop-up menus) and displaying trees (a simple layout algorithm and editor for directed acyclic graphs). The structure editor is not based on the Interlisp-D structure editor (DEDIT), which is hand crafted for the structure of Interlisp-D. Instead it was built using the programmable features of the text editor (TEDIT), which serves as a base 'virtual editing machine.'

Figure 3 shows the basic components of the system (See Nørmark, 1987b for a tutorial and overview).

The core provides basic facilities for building and manipulating structures, along with the language specification facilities needed to organize the language-specific information used by the other levels. Around this core there are a collection of generic tools, which make use of the language specification to provide structured editing, storage, parsing, etc. Finally, around these one can build language-specific tools, which take advantage of generic tools to perform their work.

3.1 The Core

At the core are the primitives for building, modifying, and accessing AST structures [Duran et al., 1987]. This core is built following conventions of data abstraction (though not using an object-oriented language) so that changes to the underlying implementations can be made without affecting other code. It includes specialized handling of AST structures used to represent lists.

The representations are designed with incremental change and mixed structures in mind. The label associated with a tree node encodes three pieces of information: a language name, a phylum name, and a version identifier. Therefore each piece of the structure carries within it sufficient information to update it appropriately in the face of language changes. This can be done incrementally (as the piece of structure is needed for some operation) or through transformations operating on a whole tree. In the current implementation, language versions operate as a whole. In the one being developed, version identifiers will be associated with individual phyla.

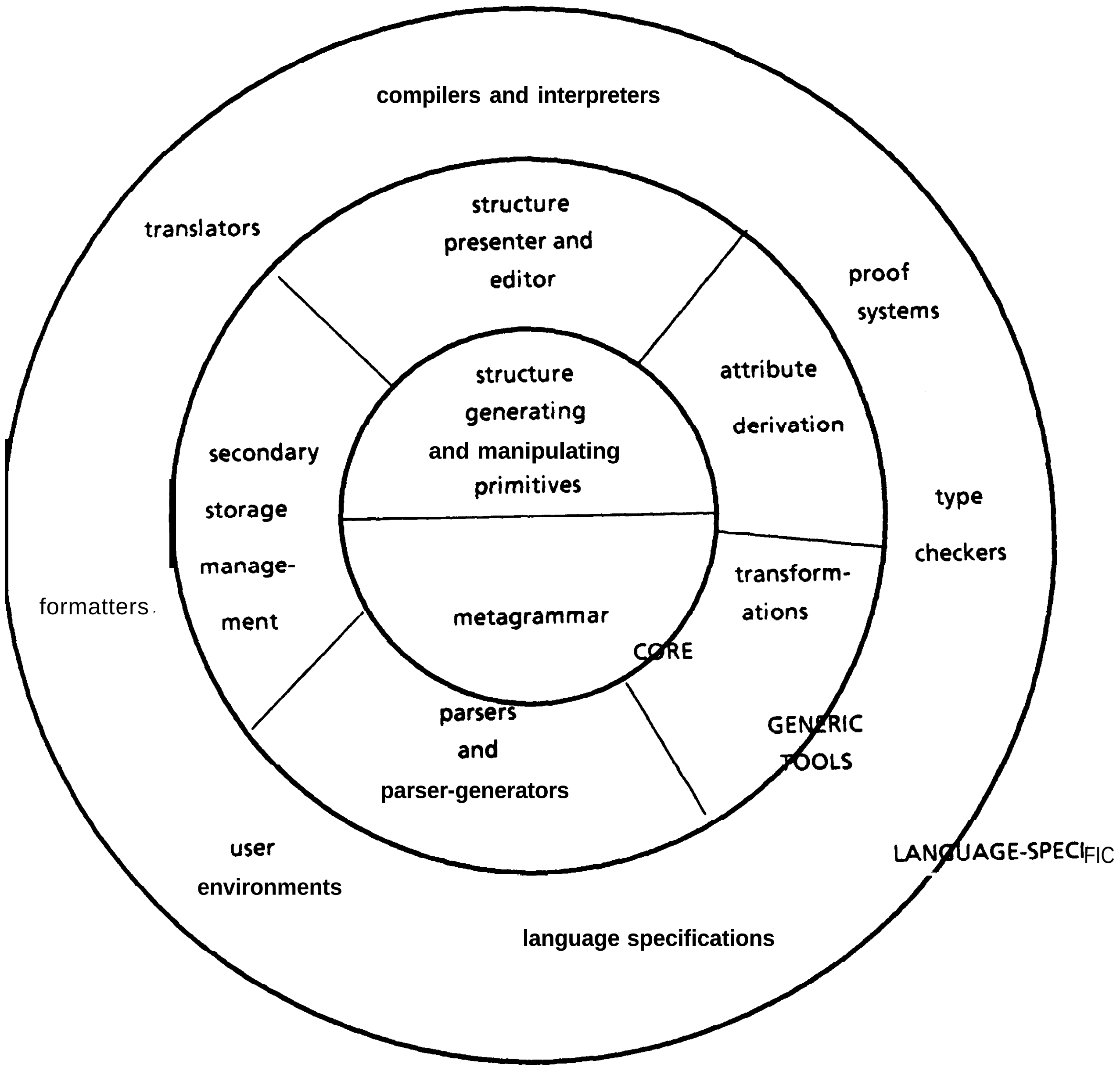

Figure 3: Components of the Muir environment.

That is, when a change is made, it updates the version of the phylum, and therefore any part of a structure not using that phylum will remain up-to-date and not need changing.

The language specification information includes the core grammar and presentation rules, attribute declarations, and facilities for associating other kinds of information with the phyla in a language (e.g., procedures designed to perform some special analysis or transformation). At the moment, our facilities for deriving this information from the structure representing the language specification (in the metalanguage) are batch-mode, operating on the whole tree. In a number of special cases, we derive information from (or 'install') a single phylum incrementally. We are working towards more incrementalism, so that the language designer can make small changes and continue working without a long compilation cycle.

3.2 Structure Editors

The largest and most important of the generic tools are the structure editors. The bulk of a user's interaction with the environment is through windows on the screen that display an AST structure (in any of the forms provided in presentation schemes) and allow the user to operate on it by selecting substructures and activating ***edit operations*** selected from a menu (which contains only those operations appropriate to the selected structure) or through keyboard commands. Any number of presentation windows can be active on the screen at once. The window placement, activation and management (e.g., scrolling, resizing and moving windows, shrinking a window to a small icon and re-expanding it on demand) are all handled by Interlisp-D.

We have adopted a very general notion of edit, operation which includes the following:

1. **Generic environment manipulating operations.** Any structure is appropriate for selecting an operation that changes the environment state. These include opening a new window with a presentation of that structure (or a copy of it) in a specified presentation scheme, writing it to a file or replacing it with one read from a file, etc. Some cross-linguistic operations are primarily useful in developing (debugging) language specifications. For example there is a generic operator that when applied to any AST node will open a new window displaying the specification (in the metalanguage) of the identification phylum for the node.

2. **Generic structure changing operations.** Some structure operations are generic, not depending on the phylum of the selected node. Some of these have standard effects (e.g., 'Copy,' and the 'Reduce' operator which replaces a node with an unexpanded node of the appropriate type), others make use of language-specification information (e.g., the 'Template' operator, which replaces an unexpanded node with a template that has been declared as the standard form for nodes of that phylum), and others are applicable only to a subclass of nodes (e.g. the list operators which apply only to list-element nodes).

3. **Simple (automatically derived) structure changing operations.** These make use of the information in the language specification to provide basic operations. For example, when an unexpanded node is' selected, the menu includes all of the operators associated with terminal phyla that are descendants of the choice phylum in the hierarchy. Selecting one of these replaces the unexpanded node with one based on the operator.

4. **Programmed operations.** For a given phylum, the language designer may want to specify specialized edit operations suited to the language. For example, a Pascal language specification may include an 'Embed statement' operation which will replace a statement with a 'begin. . . end' block containing that statement. An inverse 'Unembed block' operator might replace a block containing just a single statement with the statement. Muir provides a generalized pattern match/replace formalism that, can be used specify such operations without explicit manipulation of the Interlisp-D data structures (see [Nørmark, 1987a]). Some programmed operators affect the environment rather than the structure. For example, there could be an operator associated with a procedure-call node that displayed the definition of the procedure named in it, or one associated with a procedure declaration node that displayed a list of all the modules in which that procedure is used. At the moment these need to be done on a one-by-one basis, but there are obviously some standard patterns (e.g., cross-indexing of declarations and uses) which could be generalized and driven by declarations in the language specification rather than by specialized code.

Edit operations are used at all levels, including structure editing in the small and things that would normally be thought of as directory operations (e.g., adding, removing and deleting modules or whole language specifications from a global environment tree). In general, selection of the operand structure is done with the mouse, but there are also keyboard commands for common cases where operations are mixed with typing (e.g., select the next unexpanded node). Currently the facilities for editing tree diagram presentations are much more limited than those for text presentations.

Incremental updating algorithms are used to minimize the amount of re-presentation that needs to be done on the screen when a change is made. This is complicated by the fact that a given structure may appear simultaneously in several windows, possibly under different presentation schemes. We are letting experience determine where the efficiency problems come up, and what needs to be done to maximize visibility and accuracy of the displays, while minimizing delays.

3.3 Parsers

As mentioned above, although the parser does not play as major a role in this system as it does in a conventional environment, it is still an important component. We have developed two different parsers (neither of which is yet fully integrated into the system). The first is an LALR(1) parser based on the BOBS system [Ericksen et al., 1982]. This system checks a context-free grammar (written in a special form which includes error productions) and determines whether it is LALR(1). If so, it produces tables that can be used in an eficient parsing algorithm. If not, it indicates the problems and the user can modify the grammar accordingly. The table-generator is written in PASCAL and does not run on the workstation. There is a parser written in Interlisp-D that uses the generated tables. The current meta-grammar cannot directly produce the forms for input to the table generator, so they must be edited by hand. Also, it has not been modified for the hierarchical phylum/operator formalism (it was built during an earlier phase of development). Since we are not focussing on applications that demand efficient parsing, we have not yet brought this facility into line with the rest of the environment. If we encounter applications where mass parsing is important, we may further develop and integrate it.

The other parser is a general context-free parser, based on Kaplan's General Syntactic Processor [Kaplan, 1973]. In its existing form, GSP takes a recursive transition net grammar (weakly equivalent to a CFG) and parses in polynomial time (proportional to n2 if the grammar is unambiguous). We have integrated it into our system by adding a component that translates from our hierarchical form into the corresponding networks, and a component that translates the LISP forms produced by GSP into AST structures consistent with our grammar (see Figure 2 above for the differences).

When the parser is fully integrated it will have several properties:

1. **It will be called by a standard edit operation on any AST node.** That is, the selected node will be presented (if already expanded) in a text window, and the user will edit the text (or enter it from scratch for an unexpanded node). The parser will then parse under the appropriate category (not necessarily the distinguished symbol of the grammar) and replace the AST node with the resulting structure. If the parse fails, the user will be returned to the text editor with the faulty text and an indication of the problem.

2. **It will be able to accept special markers for unexpanded nodes.** For example, the parser working with a Pascal language specification might accept a string like 'if `x=3+(@Expression/2*units)` then y := x-6', in which the character sequence '@Expression' will produce an unexpanded node in the AST with choice phylum 'Expression.'

3. **It will be able to translate from the generalized presentation rule formalism into appropriate rules and reserved word list.** Currently it only works with the simplified presentation rule formalism, which does not allow for font changes, conditional presentation, etc.

3.4 Storage Management

If the environment needed only to deal with a single user operating in a continuously available core image (in Interlisp terms, a 'sysout') on a large enough machine, there would be no need for storage management. Interlisp-D provides a virtual 24-bit address space with automatic memory allocation and paging. This is more than adequate for our envisioned uses. But of course the real world differs from this. We need to provide for sharing of structures between users on different machines in a networked environment. Also, there need to be ways to create well-modularized stable storage structures that can encode subsets of the AST structures produced in an environment, to be read back later into other instantiations ('loadups') of MUIR.

Our initial plans called for a quite general system that would make it possible to operate in a heterogeneous environment with a variety of file servers, shared among a. community of workers, robust in the face of relatively frequent and unpredictable system failures. In particular, it needs to deal with coordinating shared files and local files (on the individual workstations), allowing individual work to continue when the shared resource is unavailable. After a fairly comprehensive initial design was developed, work was postponed in the interests of developing other aspects of the environment . At the moment, the underlying Interlisp-D file system is used in a fairly direct way. There is an operation that writes an AST structure into a file (specified by server, directory and file name), and one that reads the corresponding structure back in. All redundancy, inter-server copying, access control, etc. are simply inherited from the Interlisp-D environment (e.g., the COPYFILE function which does some format conversion) and the servers themselves (each of which, for example, has its own access control scheme). Since we have not yet attempted large shared applications, this has been sufficient.

3.5 Attribute Evaluation

Attribute manipulation (See Landgrebe, 1987) is one of the generic tools, not a part of the core. An AST need not have a consistent solution to its attribute equations in order to be treated as well formed. Satisfaction of an attribute set may be required by various language-specific tools. Solution is done for an entire AST at one time, not incrementally as structural changes are made. This means that in generic tools (e.g, the structure editor) attribute constraints are not considered. This is a limitation, and a possible direction for further research (as in [Reps, 19841).

Computation of attributes is currently done in a demand fashion which allows the value of a given attribute at a particular AST node to depend both on values of its descendants and of its parent. A list is kept of attributes whose value depends on a not-yet-available value, and whenever an attribute is computed, this list is checked for values that have become computable. This very general approach does not lead to the most efficient attribute computations, but was selected in line with our general strategy of providing a maximally flexible prototype. A language designer might well design a particular grammar with a more limited version of attribute calculation (e.g., using only synthesized attributes, or using both synthesized and inherited attributes but not allowing values of one to depend on the other), with the expectation that after the language stabilized, an environment could be built that took advantage of this restriction to do things more efficiently.

3.6 Transformations

As mentioned in Section 3.2, the edit operations provided by the structure editor are implemented using a general transformation formalism. This formalism was designed for extended uses of transformations, such as the translation of a program from one language to another, or from an earlier to a later version of the same language, or the further refinement of programs through transformation. This is particularly important in working with an evolving language. As the language changes, structures that were previously created are no longer well-formed with respect to the later version. On the other hand, one should not have to recreate them from scratch. Facilities are being provided to 'update' structures from one language version to another, with as much automation as possible. Many changes (e.g., adding new phyla) require no transformations to existing structures. Others (e.g., renamings, or the addition of elements to an operator) require transformations that can be automated. Others (e.g., replacing a phylum with two distinct new ones) will require manual updating, but the system can locate the appropriate places and give the user interactive choices of what to do. These same possibilitiesapply to other translation activities, such as translating a program from one language to another.

The basic transformation primitives use a pattern-match and replace form, and automate a number of simple transformations. The transformation component and some experimental uses of it are described in [Nørmark, 1987a].

4 Applications

In the process of developing Muir we have been motivated and guided by specific languages we wanted to work with. We are still in a stage of active development and do not have any fully-developed language-specific environments. The following are the major implementations to date:

4.1 Aleph

The Muir system was initiated as part of developing the Aleph system specification language [Winograd, forthcoming]. After developing a series of prototype environments specialized for Aleph, it became clear that it was worth the effort to do a more general environment, for which Aleph would be just one application. Many of the Muir facilities were initially designed to handle the features of Aleph, and were tested on various versions of it. The BOBS parser was applied to versions of Aleph and it was the major example for the first prototype of the structure editor. No language specification has been written yet for a full version of Aleph, awaiting the documentation of a stable working version of that language.

4.2 Pascal/Modula

As an extended example for developing theoretical work in program transformation, language specifications for Pascal and Modula were written, structures (programs) were created and transformations (translations from one language to the other) have been done. This work is described more fully in [Nørmark, 1987a] .

4.3 MetaGrammar: The Muir Language Specification Language

The most heavily used language specification developed so far is the specification of the specification language itself. This was done in a bootstrapping fashion, in which a handcrafted version was produced, which could in turn be used to bring the language, development tools to bear on the specification language itself. Currently we are operating with a fully bootstrapped MetaGrammar [Holstege, 1987], in which the language-designer can use the structure editor, storage mechanisms, attribute computations, etc. in defining a language. It has several interesting features:

1. **The standard presentation rules use a combination of tree diagrams and textual forms.** The higher level structure (e.g., the phylum hierarchy) appears as trees, while the detailed structure (e.g., the operators and presentation rules) are represented by formatted text.

2. **It provides special mechanisms for defining list and tree structures.** The basic AST format does not handle lists as special data structures, but represents them as right-branching trees. The set of relevant phyla and operators (e.g., for lists, empty lists, etc.) are generated from a higher level 'List-phylum' specification provided for in the MetaGrammar. Similarly, trees with labelled nodes (e.g., as used in a directory structure) have specialized specification forms, but create AST nodes with standard structure.

3. **It provides for 'compiling' the language specification for efficiency.** The form that is most convenient for a language designer is often not the most efficient for use by generic tools like the structure editor. For example, the set of terminal phyla associated with a categorial phylum needs to be known every time a menu of edit operations is created for that phylum, and must be computed by a search in the phylum hierarchy. This search is done once and the result stored (as a property) for use by the editor. Currently these compilations are done by specially written Interlisp-D code, but in the near future we hope to use the attribute equation and evaluation facility to do most of them in a structured way.

5 Status and Conclusions

Muir is in an active state of development, not yet ready for general use. Many of the facilities were described above as currently being implemented or scheduled for future work. Our plans involve developing specifications for a number of sample languages (including some from logic and mathematics, as well as programming and specification) in order to reveal the shortcomings and point to the best directions for development. In addition, a number of obvious efficiency issues must be addressed before the performance will be acceptable to any but highly-motivated users.

It is too early to state confidently that Muir will be a practical tool for language designers, but our experience so far (including our own use of it as a tool for writing and modifying language specifications) confirms our belief that it has great promise, both as a concept and as a particular implement ation.

Appendix: The Muir Grammar Formalism

This is intended as an explanatory introduction. For a more thorough and precise definition, including multi-formalism grammars and issues of transformation between grammars, see [Nørmark, 1987a].

We are concerned with a grammar that is used to define a set of possible **AST structures:**

- An *AST* ***structure*** is a tree structure made up of *AST* Nodes. It differs from a standard tree structure in that the ordering of branches within a node is not considered, and the label of a node is made up of two symbols, called the ***Identification phylum label,*** and ***Child identifier.***

- In addition there is a function from nodes of an AST structure to ***Property lists,*** described further below.

A grammar is represented as a graph structure whose elements are ***Phyla,*** linked by a relation of ***Subphylum*** (which is reflexive-a phylum is a subphylum of itself). It is a directed acyclic graph (which can be thought of as a partial ordering, or as a tree in which a node can have multiple parents).

- There is l-l function mapping every phylum in the grammar onto a token called its name (no two phyla share a name).

- There is a function whose domain is the set of terminal phyla (a phylum whose only subphylum is itself), and whose range is a set of ***Structural*** *descriptions.*[1]

- A structural description consists of an unordered set of ordered pairs of names (the ***Child name*** and the ***Choice phylum name).*** No two pairs in a structural description have the same child name.[2]

An AST is well-formed with respect to a grammar, if and only if:

1. For every node, the identification phylum label names a phylum in the grammar (which we can call the 'identification phylum' for the node.

[1] **I have used the term 'structural description' in place of 'operator' in order to avoid debates with regard to previous versions of this and other systems.**

[2] Note: **If we were to use initial segments of the natural numbers as sets of child names, the structure would be equivalent to using ordered- trees with no child names. We have not done this in order to emphasize the fact that ordering of elements belongs to the presentation rules, not the abstract grammar.**

2. For every node that has children, there is a one-one correspondence between the node's children and the elements of the structural description associated with the node's identification phylum, such that:

   (a) The child name of the SD pair is equal to the child identifier of the child

   (b) The identification phylum name of the child names a subphylum of the phylum named by the choice phylum name of the SD pair.[3]

3. The root node of the tree has the distinguished token 'ROOT' as its child name[4]

[3]Childless nodes include those corresponding to CFG terminal symbols, or to other nodes that are unexpanded but belong to phyla that could include expansions with children. Note that a well-formed AST may include unexpanded nodes

[4]Note that this does not specify the phyla of the root, except to the degree they are constrained by the above. A legal AST structure can have any phylum at its root.

References

SESPDE stands for ***Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,*** issued as ***Software Engineering Notes*** 9:3 and ***SIGPLAN Notices*** 19:5, May 1984

Donzeau-Gouge, Veronique, Gerard Huet, Gilles Kahn, Bernard Lang. 1980. Programming Environments Based on Structured Editors: The Mentor Experience. Inria, ***Rapport de Récherches***, no. 26. July.

Donzeau-Gouge, Veronique, Gilles Kahn, Bernard Lang, B. Melese. 1984. Document Structure and Modulariy in Mentor. ***SESPDE,*** 141-148.

Duran, Raul, Mary Holstege, Kurt Nørmark and Liam Peyton. 1987. Muir Tree Documentation. CSLI Informal Note #87-6.

Eriksen, S.H., B.B. Jensen, B.B. Kristensen, and O.L. Madsen. 1982. The BOBS-System. Aarhus University, Computer Science Department PB-71, 3rd Ed.

Hansen, Wilfred J. 1971. Creation of Hierarchic Text with a Computer Display. Ph.D. dissertation; Stanford University.

Holstege, Mary. 1987. The Muir MetaGrammar. CSLI Informal Note #87-7.

Kaplan, Ronald. 1973. A Multi-processing Approach to Natural Language. ***Proc. National Computer Conference.*** Montvale, New Jersey: AFIPS Press.

Knuth, Donald. 1968. Semantics of Context-free Languages. ***Mathematical System Theory Journal,*** 127-145.

Landgrebe, Birgit. 1987. The Work with Attributes in the Muir Environment. CSLI Informal Note #87-8.

Linton, Mark. 1984. Implementing Relational Views of Programs. ***SESPDE,*** 132-140.

Medina-Mora, Raúl. 1982. Syntax-directed Editing: Towards Integrated Programming Environments. Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, March.

Nørmark, Kurt. 1987. Transformations and Abstract Presentations in a Structure-oriented Editing Environment. Dissertation Aarhus University, CSLI Informal Note #87-9.

Notkin, David. 1985. The Gandalf Project. ***The Journal** of **Systems and Software,*** 5:2 (May).

Peyton, Liam. 1987. Presentation in a Language Design Environment. CSLI Informal Note #87-10.

Reps, Thomas. 1984. ***Generating Language-based Environments.*** MIT Press.

Reps, Thomas and Tim Teitelbaum. 1984. The Synthesizer Generator. ***SESPSDE, 42-48.***

Teitelbaum, Tim and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. ***CA CM*** 24:9 (September) 563-573.

Teitelman, Warren. 1969. Toward a Programming Laboratory. ***International Joint Conference on Artificial Intelligence,*** Washington, May, l-8.

Wile, David. 1982. POPART: Producer of Parsers and Related Tools, System Builder's Manual. USC/ISI TM-82-21.

Winograd, Terry. forthcoming. Aleph: A System Specification Language. CSLI report.