Syntactic and Semantic Checking in Language-Based Editing Systems

Robert Alan Ballance

Computer Science Division—EECS
University of California, Berkeley
Berkeley, California 94720

December 1989

Dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate Division of
the University of California, Berkeley

Report No. UCB/CSD 89/548

This research was supported in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 (monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089) and Arpa Order No. 6379 (monitored by Space and Naval Warfare Systems Command under Contract No. N00039-88-C-0292), by IBM under IBM Research Contract No. 564516, and by a State of California MICRO Fellowship.

Syntactic and Semantic Checking in Language-Based Editing Systems

Robert Alan Ballance

Abstract

Language-based editors and browsers allow users to create, browse, and modify structured documents in terms of the syntax and semantics of the underlying language. The feasibility of editors which fully combine text- and structure-oriented operations has already been shown. However, prior investigations show limitations which are addressed by the research reported here. First, earlier systems have either constrained text-oriented editing or have forced the user to manipulate a structure based on the needs of an incremental parser rather than on the abstract syntax of the language. Second, specifications of contextual constraints generally have been difficult to write and to understand. In the quest for efficiency, the logical form of the specification has been obscured by concentration on low-level details. Third, the approach to checking contextual constraints has often been oriented toward translation rather than browsing. The information gathered during analysis is made available only to the analyzer, and not shared by other tools.

Grammatical abstraction and *logical constraint grammars* are new approaches to specifying and enforcing the syntactic and static-semantic constraints of a language within a language-based editor. Grammatical abstraction defines a formal correspondence between the concrete (parsing) syntax of the language and the abstract syntax of the language as viewed by a user of the system. This correspondence allows a decorated abstract syntax tree to be used during incremental LR parsing.

Logical constraint grammars couple logic programming with consistency maintenance to describe and enforce the static-semantic constraints of a language. The information required for incremental consistency maintenance is derived directly from the description of the semantic constraints. The author of a language description is shielded from most of the details of information flow and storage management. Information gathered during enforcement is retained in a logic database accessible by other tools.

Pan is an experimental language-based editing and browsing system that uses grammatical abstraction and logical constraint grammars. Experience with *Pan* confirms the utility of these approaches.

Contents

List of Figures

List of Tables

Acknowledgements

Pan, COLANDER, and the work presented here have benefited immensely from the comments, questions, critiques, and suggestions of many individuals. Fellow graduate students—Ricki Blau, Bob Corbett, Charlie Farnum, Phil Garrison, Robert Henry, Peter Kessler, Jim Larus, Kirk McKusick, Bill Maddox, Eduardo Pelegrí-Llopart, Dain Samples, Polle Zellweger, Ben Zorn—were sources of inspiration and friendship. Professors Michael Harrison, Richard Fateman, and Alvin Despain, while not directly involved in this work, were all most helpful.

At the University of New Mexico, where this acknowledgement is being written, I want to thank Professors Ed Angel, Jack Brayer, George Luger, Barney Maccabe, and Brian Smith for their help and support.

Of those on the borders of academe, Steve Muchnick, Michael Mahon, and Terry Miller deserve special thanks.

The members of the PIPER community have all made major contributions to the design and implementation of *Pan*, LADLE, and COLANDER. Michael Van De Vanter was constant as a colleague, sounding board, critic, collaborator, source-code manager, juggling partner, and friend. Jacob Butcher implemented LADLE, helped to develop the theory of grammatical abstraction, and has labored continually improve the system. Chris Black forced me to consider semantic support for pretty-printing. Darrin Lane ported *Pan* to Common Lisp.

The members of my committee: Susan Graham, Paul Hilfinger, and Lucien LeCam have helped to make this work as complete and as readable as it presently is. Any remaining flaws in the work are my own. Susan Graham, my research advisor, is to be thanked especially for her guidance, patience, and support.

Finally, words cannot express how much help my wife, Kathy, and my daughter, Elizabeth, have been in this work. I'll have to take them on vacation, instead.

Chapter 1

Introduction

> Looking back it is hard to see now why the ultimate designs were not visualized sooner. However, accepted practices are generally difficult to change, and it must be remembered that, owing to the lack of more accurate data, many of the early designs were the result of trial and error. As test data and operating experience accumulated, the designers proceeded with greater confidence.
>
> Alfred W. Bruce
> *The Steam Locomotive in America*

Exploration of the design space, experimentation with new materials, investigation of alternatives, the quest for engineering principles— these characterize a maturing yet not mature discipline. By the mid 1800s, the possibility and promise of steam-powered locomotion had been demonstrated. Through trial and error, by building, running, and analyzing locomotives, the design of a steam locomotive became a well-understood engineering task. But in the beginning, it was different. There were few rules, little data, and many alternatives seemed worthy of attention.

Designers of language-based software tools find themselves in a similar situation today. There are few rules, and the data from early investigations provides only rough guidance. Many alternatives seem worthy of consideration. Numerous tools, document editors especially, have been built. Some are in use. But the design rules are still open. Building language-based software tools requires engineering and insight, trial and error, patience and skill. The research reported here documents another foray into the design and implementation of language-based editors and browsers.

Language-based software tools use knowledge of the languages being manipulated to provide specialized operations and error checking. One class of language-based tools, language-based document editors and browsers, allows a user to create, maintain, and examine documents interactively. The editor or browser itself is a program or collection of programs that mediates between the user and one or more documents. In this role, it provides controlled access to the document while communicating with the user about the state of the document. This position—between document and user—enables an editor to maintain and analyze sequences of user actions and their effects on the document; precisely the situation needed for the front end to an integrated development environment.

Henceforth, the term **editor** will denote a computer program for altering, manipulating, and viewing documents; the term **document** will denote the objects accessed through an editor and manipulated by it; and the term **user** will denote the person or process that is accessing the document. This use of "editor" subsumes both editing and browsing capabilities.

Programming languages are described in terms of their syntax and their semantics. The **syntax** of a programming language defines the *form* of a program. The **semantics** of a programming language defines the *meaning* of a program.

The semantics of a programming language can be divided into two portions: aspects that can be verified independently of program execution and aspects that may require program execution in order to be calculated. These two portions of the semantics of language are called the **static semantics** and the **dynamic semantics** respectively. The static semantics describes textual aspects of the semantics while the dynamic semantics describes temporal behavior.

For example, consider the assignment statement ⟦ X := Y + 1.0 ⟧[1]. The dynamic meaning of this statement might be that "the current value of ⟦ Y ⟧ is added to the constant ⟦ 1.0 ⟧ and the result is stored in the location denoted by ⟦ X ⟧". The static meaning of this expression might be that "If ⟦ X ⟧ and ⟦ Y ⟧ are declared in the current environment, and the type of ⟦ Y ⟧ is either **integer** or **real**, then the type of the expression is **real**; the assignment is valid if the type of of ⟦ X ⟧ is **real**."

The static semantics of a language can be described using a collection of constraints associated with the syntactic structures of the language.

This research is directly concerned with ways to describe the syntax and static semantics of programming languages, and how to use those descriptions to provide error checking and other operations in a language-based editor. While the focus here is on programming languages the results presented can be adapted easily to other, less complex, formal notations.

Research in language-based editing of programming languages often regards the editor primarily as the front end to a compiler; the issues, problems, constraints, and solutions therefore become biased towards language translation. By focusing on the editor as a front-end to an integrated environment and by stressing support for the creation, manipulation, maintenance, and evolution of programs, a different set of solutions have emerged. Those solutions are applicable to many language-processing problems, but are tuned to the problems of textual analysis and knowledge-based interactions.

A language-based editor yields its maximum benefit when it is integrated into a larger tool set. The editor then becomes the primary interface to an integrated environment, perhaps supported by a program data base. A language-based editor serves as the front end to execution facilities (interpreters, code generators) as well as the primary tool for maintaining program sources and documentation. Using a language-based editor as a front end for execution facilities eliminates the continual reparsing of the program, yielding more rapid execution of the program-object. Because of the ability to handle both text and structure, and because user interfaces can be uniform, a language-based editor that provides both text- and structure-based editing operations is an excellent choice for the front end to

[1] The notation ⟦ ... ⟧ is to quote syntactic units.

an integrated environment.

1.1 Desiderata for Language-Based Editors

As a class, language-based program editors have been with us since at least Hansen's work in 1971 [51], yet recent papers by Lang [81] and Neal [95] question their utility. Their lack of impact stems from three sources.

First, coding a program is usually a very small component (Pressman [106] claims 1/4–1/6 of the total time) in the software development process. Used only for coding, a language-based editor reduces at best only a fraction of the cost of developing programs. Yet editing programs and other structured documents is continual as a design unfolds. An editor able to handle a variety of formal languages while presenting a uniform user interface is valuable throughout the design of a large system. Such an editor could be used for both programs and documentation, for design languages, mail, memos, and most other editing tasks.

Second, treated as a stand-alone system, a language-based editor can assist only in the task of editing. It is when an editor makes use of its position between the user and the document to act as a front-end to a more powerful integrated and knowledge-based environment that its capabilities will prove essential. As a front-end, the editor is able to gather, maintain, and manipulate information about the document as that document is being updated. The information gathered can be shared with other tools in the environment; other tools can themselves generate information useful to the editor or its users.

Third, early language-based editors provided their facilities at a high cost to the user, either by constraining the ways that a user could interact with a document or by using excessive amounts of computation or storage. Neal [95] supports this analysis in her investigation of how programmers use a syntax-directed editor:

> ... very experienced users will reject any tool that does too much for them. For instance, they will only use a tool which allows free text input but provides error checking capabilities.

Experienced users require both flexibility and high functionality in their editors.

The following characteristics are desirable in any language-based editor:

- The editor should gather and maintain information about the documents being edited and should be capable of sharing that information with the user and with other tools in the environment.
- The editor should support multiple production languages such as C, Modula-2, or Ada.
- The editor should present uniform user and programmatic interfaces for all languages that it supports.
- The editor should not impose rigid constraints on how the user may edit the document.

- The editor should shield the user from the details of the internal representation of documents.
- The editor should detect all syntactic and static semantic errors, and should recover from them.
- The editor should provide tractable specification techniques for adding new languages.
- The editor should be efficient enough to be the editor of choice in most situations.

The above order of presentation does not imply a relative order of importance to the user; users adapt remarkably well to diverse interfaces, yet begrudge any system which slows their work.

As software systems (and documents) become more complex, the burden of a developer shifts from one of actively changing the system to one of actively understanding the system. As Winograd [139] states so well: "The main activity of programming is not the origination of new independent programs, but in the integration, modification, and explanation of existing ones." If this is so, then the orientation of an integrated development environment shifts from support for the creation of new code to support for the analysis, understanding, and explication of the existing systems. Again, quoting Winograd, "The main goal of a programming system should be to provide a uniform framework for all information that now appears (if at all) in the declarations, assertions, and documentation." The editor is an essential tool in the process of gathering and maintaining such information.

The editor is the primary tool for the computer programmer. More time is spent using the editor than using any other tool. As systems become more integrated, this dominance will increase.

It is therefore desirable that a single interface be available for editing different documents written in disparate languages. Rather than creating separate tools—one (or more) per language—a single tool that handles multiple languages should be created. A language-based editor that provides unrestricted text editing can be used as a text editor, so the support provided by such an editor is no worse than one has presently. Moreover, a multilingual language-based editor must handle production languages. The practicing programmer has the most to gain from the availability of multilingual language-based editors.

Because of the concern for the practicing programmer, one must create a model of interaction allowing maximum flexibility while offering maximum safety. Flexibility comes from the availability of both textual and structural models of editing; safety from the availability of error detection and error recovery.

The last two characteristics stem from efficiency requirements: for the user of the editor, and for those who add new languages its repertoire. Defining "efficiency" in language-based editors is difficult because the editor performs most of the checking now performed by a compiler. Necessarily, this incurs both time and space overhead. With high-performance workstations now available, this cost will appear less. Demanding efficiency is essential, for the editor must not hinder creative expression.

Providing the above characteristics depends on the design of the major components of a multilingual language-based editor and on the interactions between those components. The research reported here deals primarily with two aspects of language-based editing:

incremental parsing using clean internal representations and the description and evaluation of static-semantic constraints. The solutions proposed address, in part, the observations about language-based editors presented above.

1.2 Thesis and Scope of This Research

The thesis of this work is that effective multilingual language-based editors providing both text- and structure-oriented commands while presenting the user with a uniform and comprehensible interface can be created. In pursuing this thesis, three major problem areas soon became central to the research.

First, previous editors that provided both text- and structure-oriented operations either severely constrained text-oriented editing or forced the user to manipulate a structure based on the needs of an incremental parser rather than on the abstract syntax of the language. In some editors, structure-based editing predominated, and text-oriented editing was restricted to certain contexts, such as entering expressions or identifier names. In systems that allowed text-oriented editing of arbitrary text, either the context for text-oriented editing was radically distinct from the context for structure-based editing (for example, when text-oriented editing occurs in a separate screen location, or even in a different editor coupled to the program editor), or the editor presented a structure derived directly from the parse tree maintained by an underlying incremental parser.

Editing in different contexts requires an explicit shift of the attention on the part of the user. Human users appear able to intermix text-oriented and structure-oriented views quite freely, so the need for explicitly shifting between the two views should be minimized. Support for free intermixing of text- and structure-oriented editing requires some form of incremental parsing.

But structure-oriented editing based on a full parse tree, as will be shown, has limited utility. The ability to parse textual input incrementally (locally) while retaining the smaller size and the simpler structure of an abstract syntax tree representation was the first problem to be attacked.

Second, specifications of static semantics have generally been difficult to write and to understand. In the quest for efficiency, the logical form of the specification has been obscured by concentration on low-level details. Action routines (Section 4.2.2) and attribute grammars (Section 4.2.1) both suffer from this problem: the interesting stuff is hidden in the code.

Third, the approach to checking static-semantic constraints has often been oriented toward translation rather than browsing. The information gathered during analysis is made available only to the analyzer, and not shared by other tools. As has been argued already, the language-based editor must share its information with other tools.

1.3 Overview of Dissertation

Pan[2], a multilingual language-based editing and browsing system under development at the University of California, Berkeley, demonstrates new approaches to these

[2]Why "Pan"? In the Greek pantheon, Pan is the god of trees and forests. Also, the prefix "pan-" connotes

problems.

As a front-end to an integrated development environment, *Pan* is able to analyze programs and other documents and to share the information gathered with other tools. *Pan* includes both a standard editor and mechanisms for adding new languages. It can be used to edit text and tree-structured documents; development has focused primarily on experiments in editing programming languages. *Pan* is able to verify the syntactic and static-semantic constraints imposed by a language. Chapter 2 describes the overall architecture of *Pan*.

Languages are described to *Pan* by a **language description**. The language being described is called the **target language**. The *Pan* editor uses a language description for a target language to determine whether a document is **well-formed**, that is, to determine whether the document satisfies the syntactic and static-semantic constraints required by the target language.

Figure 1.1 summarizes the relationships between target languages, documents, language descriptions, and languages for describing languages. A language description describes the syntax and semantics of some target language. The language description is itself written in a **language description language** which is based on a collection of language description techniques. The language description can be used by one or more interpreters to enforce the syntactic and static-semantic constraints of the language[3]. An interpreter for the *static* semantics of the target language executes the *dynamic* semantics of the language description language.

Two new language description techniques were developed for *Pan*. The first, *grammatical abstraction*, establishes a formal correspondence between the concrete (parsing) syntax and the abstract syntax of a language. This correspondence allows a decorated abstract syntax tree to be used as a basis for incremental LR parsing as well as the basic representation for structure editing. A separate categorization of the elements of the abstract syntax further shields the user from the details of the internal representation. LADLE [23] is a language and preprocessor for *Pan*'s language descriptions. When provided with the syntax-oriented aspects of a language, the LADLE preprocessor verifies that the relationships required by grammatical abstraction are present and then generates the language description tables that are used by *Pan*. Chapter 3 provides the details of this approach.

The second approach, that of *logical constraint grammars*, is used to specify and enforce the static-semantic constraints of a target language. Logical constraint grammars use logic programming to express contextual constraints. Goals associated with syntactic structures specify the static semantics of the target language. Structure-independent clauses establish the axioms of the semantic definition.

Information gathered during constraint checking is retained in a logic database. A consistency maintenance mechanism revises the database and (re)attempts goals as necessary. The author of a language description is shielded from most of the details of information flow and storage management. Other tools, as well as the editor and the user, have access

"applying to all"—in this instance referring to the multilingual text- and structure-oriented approach adopted by this project. Finally, since an editor is one of the most frequently used tools in a programmer's tool box, the allusion to the lowly, ubiquitous kitchen utensil seems apt.

[3]If the language description describes the dynamic semantics of the target language, a interpreter for the execution of a program could be written as well. Such interpreters are outside the scope of this research.

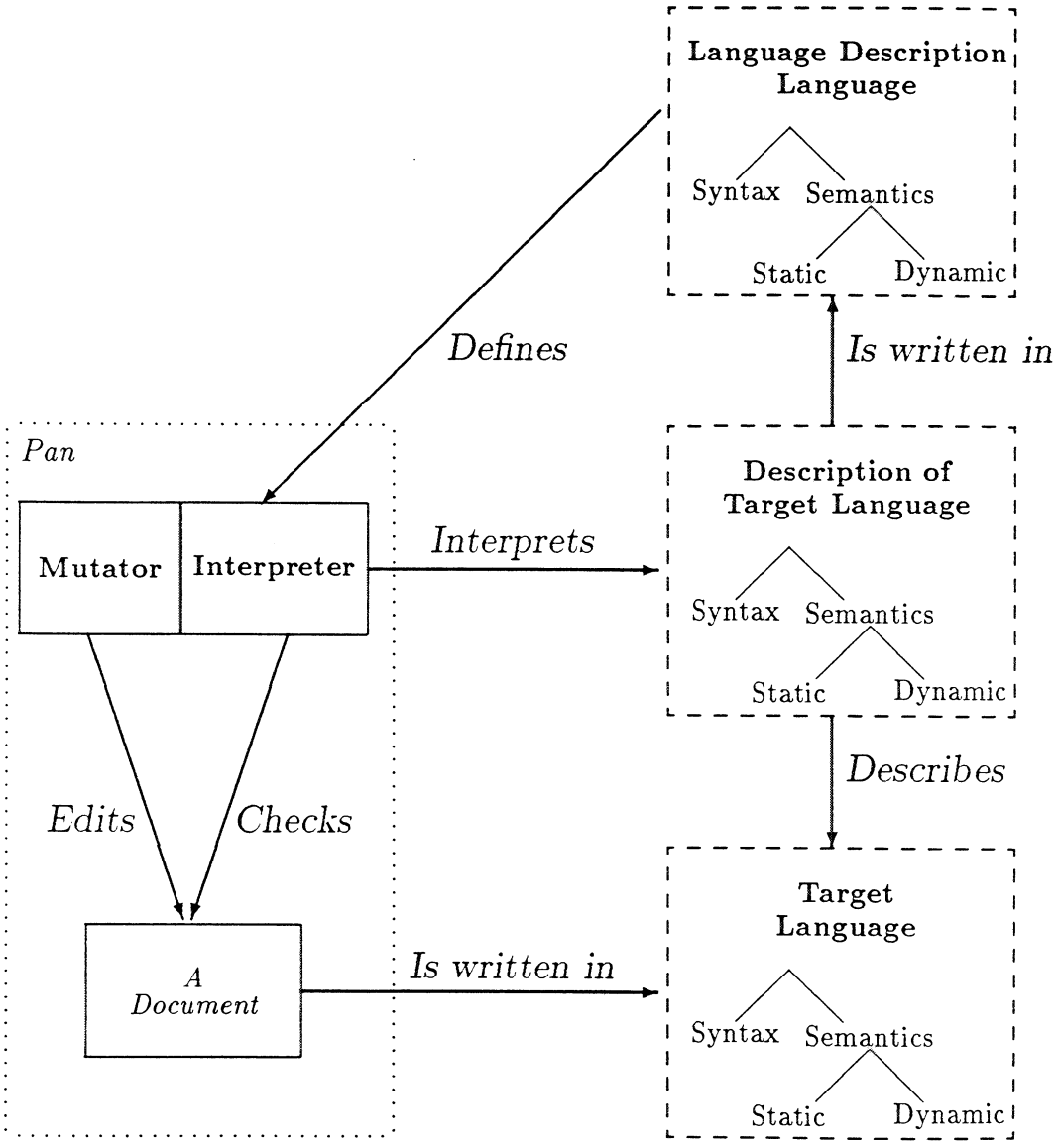


Figure 1.1: The Role of Language Descriptions

to the data in the database via a client interface. COLANDER[4] is the component of *Pan* that is used to describe and check contextual constraints. Chapters 4–7 describe this new approach.

Chapter 8 looks back on what has been learned and looks forward to the questions that have been raised by this work.

[4] COnstraint Language AND EvaluatoR

Chapter 2

The Pan I Editor

The *Pan* editor is a multi-window, mouse-based editor and browser that was developed to support ongoing research into integrated document development environments. *Pan I* [15, 16] is the currently implemented version of *Pan*. *Pan I* runs under UNIX[1] on the Sun Workstation[2] using the SUNVIEW [127] windowing system. Throughout this document, the term "*Pan*" refers to *Pan I*.

The design of *Pan II*, a new editing and browsing system using the lessons learned from *Pan I*, is now underway. Major goals include more advanced presentation and browsing facilities, a more integrated model of text and structure editing, and the ability to run as a program-driven tool. The implementation will use the X window system [122]. *Pan II* will serve as the basis for integration into a larger language, program, and document development environment.

Figure 2.1 shows the screen of a workstation running the *Pan I* editor. In *Pan*, each **buffer** contains a document that can be manipulated through one or more **viewers**. Documents are stored in files. For a preexisting document, a buffer always contains a copy of the file. The "base buffer", appearing in the upper left hand corner, lists the files being edited. Also visible are a viewer onto a buffer of text ("intro.tex"), and a viewer onto a buffer containing a Modula-2 program. Finally, a help viewer appears in the upper right-hand corner showing the key bindings in effect for buffer "BQueue.mod".

This chapter presents the issues, techniques, and architecture[3] adopted for the *Pan* editor. It begins with a review of the editing systems that most influenced the design *Pan*. The design and architecture of *Pan* is then reviewed, after which several thorny issues concerning interrelationships among the components of *Pan* are discussed.

2.1 Classification of Language-Based Editors

The terms "text-editor", "structure-oriented editor", "syntax-directed editor", and "syntax-recognizing editor" characterize different approaches to language-based editing.

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

[2] Sun Workstation, and SunView are registered trademarks of Sun Microsystems, Inc.

[3] The architecture of *Pan I*, Version 2.3, is described more completely in a related technical report [16].

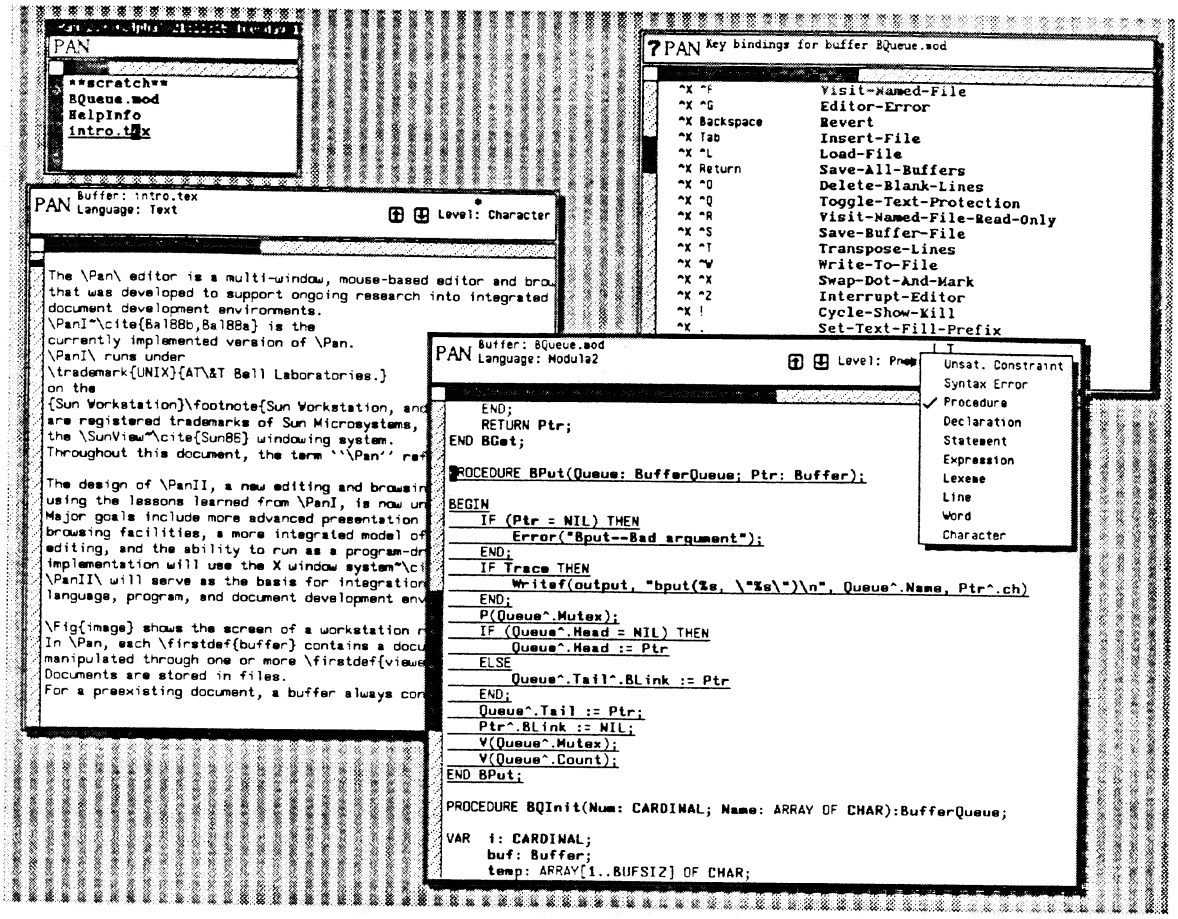


Figure 2.1: Screen Image of *Pan*

Each term denotes a metaphor that underlies the user interface provided by the editor. Figure 2.2 summarizes the classification used here. Of the classifications, syntax-recognizing editors are the most useful for practical programming environments. *Pan* is a multilingual syntax-recognizing editor that provides unrestricted text-oriented operations.

A **text-oriented editor** operates on a document modeled both as a stream of characters and as a two-dimensional plane of characters. (The coordinates in this plane are commonly called "lines" and "characters within a line".) Users are free to operate on any character at any time. While groups of characters such as words or lines might be accorded special status and operations, no special structural constraints are imposed on the document. Most text editors have the interesting characteristic of allowing a user to freely intermix the one- and two-dimensional models of text. Language-based text editors provide commands that operate on textually recognizable elements of a language, but they do not provide structure-oriented commands or complete syntactic or semantic error checking. EMACS [124, 125], and the Z editor [142] are text-oriented language-based editors.

A disadvantage to this approach is that the editor operates only on the textual representation of a program. Complete error checking cannot be performed until parsers

- Text-Oriented
- Structure
  - Pure (uninterpreted structure)
  - Syntax-Directed
    - Pure
    - Restricted Text
  - Syntax-Recognizing
    - Restricted Text
    - Unrestricted Text

Figure 2.2: Classification of Language-Based Editors

and semantic analyzers are explicitly programmed into the system.

A **structure editor** presents the document as having a definite internal structure that can be manipulated by the user. The operations supported by a structure editor are modeled as operations upon the underlying structure. Most often, the document is tree-structured, with operations defined on subtrees. Outline processors are structure editors for tree structures. Structure editors may or may not interpret the structures that they edit. When they do attribute meaning to the structures, they are often called syntax-directed editors.

A **syntax-directed editor** is a structure editor which requires that the document be syntactically correct at all times: editing operations must "follow" the syntax of the language. As in a pure structure editor, the user can add new material to a document only at those points where it can be successfully grafted onto the existing structure. The Cornell Program Synthesizer [130] is a familiar example of a syntax-directed editor. The term "structure-oriented" [101] is sometimes used in place of "syntax-directed".

Syntax-directed language-based editors sometimes enforce contextual constraints as well as maintain syntactic consistency. Some syntax-directed editors provide limited forms of text-editing beyond that strictly required for program entry. For instance, in the GANDALF [70] editors and the Cornell Program Synthesizer [130], expressions may be entered as text and then parsed for correctness. A syntax-directed editor that can parse from text expressions or other small syntactic units provides restricted text editing.

Structure and syntax-directed editors generally offer only a top-down model of development; but recent versions offer textual operations in selected contexts. While the syntax-directed development style is useful in some cases, (e.g., in environments for novice programmers or in environments for constructing mathematical proofs [28]) the practicing programmer has more general needs.

A **syntax-recognizing editor** [22] uses syntactic and semantic information to check for correctness without necessarily demanding the consistency constraints of a syntax-directed editor. Syntax-recognizing editors provide structural operations, but may provide text-oriented editing as well. The text-oriented editing provided by a syntax-recognizing

Structure-Oriented Commands	Text-Oriented Commands
1. Copy the statement to a safe place	1. Select the statement
2. Delete the statement	2. Insert the keyword **begin** before the statement
3. Insert a compound statement in place of the original statement	3. Insert the keyword **end** after the statement
4. Reinsert the original statement at the subtree embedded within the compound statement	

Figure 2.3: Changing a Simple Statement to a Compound Statement

editor may be restricted or unrestricted. Restricted text editing includes the case where the editing is confined to specific subtrees. Normally, in this case, the user must explicitly select the structures to be edited textually prior to performing the editing. Syntax-directed editors with restricted text-editing facilities and syntax-recognizing editors with restricted text-editing facilities are quite similar.

A **multilingual** editor is one that supports multiple languages.

For obvious reasons, a pure structure editor is seldom implemented. In fact, almost all editors are hybrids. Text editors are extended to handle language issues; structure editors are extended to handle textual input; syntax-directed editors are extended to allow inconsistency. Syntax-recognizing editors that provide unrestricted text editing form the most general class of editors to date. For the practicing programmer, this has important implications.

Suppose one is editing a Pascal program and wishes to change a simple statement to a compound statement. Figure 2.3 contrasts actions in a pure structure or syntax-directed editor with those of a language-based editor that provides unrestricted text-oriented operations. A purely structure or syntax-directed editor would require a sequence like that shown on the left-hand side of Figure 2.3. (Of course, the structure editor might have a "make this statement a compound statement" command. The availability of many such commands complicates the user's world, as well as the editor designer's world.) The actions presented on the right of Figure 2.3 are those that would be used in a text-oriented editor like EMACS [124, 125] or in any language-based editor that supports unrestricted text editing. Immediately after the insertion of the **begin** keyword at step 2, the program is syntactically inconsistent, a condition disallowed by a syntax-directed editor.

Documents are rarely maintained in the top-down manner implied by structure editors. Usually, the user's task is modification, not creation. Often, the program source is usefully viewed as text; for instance when making corrections (misspelled identifiers, incorrectly ordered parameter lists), writing comments, or searching for patterns. It should be left to the user to determine the style of interaction most desirable in a given context. The structure-oriented model of top-down, successive elaboration of templates corresponding to language constructs should be available; not mandatory.

Syntactic and semantic checking are even more necessary than textual manipulations. The strict constraints imposed by a syntax-directed editor support novice users and

facilitate the task of checking correctness of the program-object. With reasonable incremental parsing and semantic checking algorithms, those constraints can be loosened without losing the ability to perform syntactic and semantic checks.

Much has been written about whether an editor should support text operations, or only structure operations [26, 123, 135, 142]. This controversy is misdirected. In actuality, programs are generally more complex objects than advocates of either approach admit. Programs are most usefully regarded as richly interconnected, overlapping webs of information. The primary linguistic structure manipulated by a language-based editor is partly an artifact of language processing. Editing locally, the linguistic structure is important; browsing globally, the linguistic structure may not be relevant. One promising research direction is to use the semantic information collected and maintained by *Pan* to provide "semantics-directed" browsing of the other structures in a program [132]. The next sections introduce a number of text-oriented, syntax-directed, and syntax-recognizing editors that influenced *Pan*.

2.2 Related Editing Systems

This section presents brief descriptions of the editing systems that have had major effects on the development of *Pan*. Many of these examples are used in later chapters when contrasting their approaches to *Pan*'s.

2.2.1 Text-Oriented Editors

EMACS [124, 125] is an extensible, customizable display editor that operates on text. Because EMACS supports an internal programming language, special commands can be implemented to provide language-specific operations. For instance, a command can be written to move the cursor from S-expression to S-expression in LISP, thereby providing a limited form of syntax-recognition. Each language can have its own special "major mode" that defines a collection of language-specific definitions and key bindings. EMACS is customizable, so that roughly equivalent actions for different languages can be made to appear as the same command.

2.2.2 Structure-Oriented and Syntax-Directed Editors

Most structure-oriented editors are description-driven. New editors for specific languages are created by writing a language description. The description is processed to create a set of tables and code that can be used to generate a new editor. Thus, an editor generator contains an editor kernel, a way to describe languages, and a preprocessor for language descriptions.

EMILY [51, 52] was the one of the first multilingual syntax-directed editors. As such, it showed the potential of such systems long before most users had sufficient computational support to use such an editor. EMILY was description-driven, using a context-free grammar to define the syntax of the language. No semantic checking was provided.

ALOE [90, 91] was an early structure-editor generator. It is based on a language-independent editor that uses tables created from language descriptions. Non-syntactic con-

straints are enforced by action routines: hand-coded procedures associated with structural elements. Whenever an editing action affects a particular structure, the relevant action routines are invoked. ALOE-generated editors figured prominently in the work of the GANDALF project [101, 102]—a study in the automatic generation of software development environments.

MENTOR [38, 39] is a syntax-directed editor developed at INRIA as a component of an interactive programming environment. It manipulates abstract syntax trees during editing. The basic editing model is a structural one: tree nodes are designated, then manipulated. Programs can be created from scratch in the editor, or can be created using a text editor and then read into the system.

New languages descriptions for MENTOR are written using the language-definition language METAL [67]. METAL provides the means to describe the syntax of a language, together with transformation rules that map the concrete syntax of the language into abstract syntax trees. Objects can be built either top-down, using templates or by entering text that is then parsed and transformed into abstract syntax trees.

CENTAUR [20] has recently been developed at INRIA as a successor to MENTOR. Like MENTOR, CENTAUR is a primarily structure-oriented editor generator with excellent facilities for language description and document representation. The kernel of CENTAUR has two specialized components: an abstract machine for manipulating trees and a logical machine based on PROLOG together with a semantic specification method called "Natural semantics" [66]. CENTAUR can also manipulate several kinds of graphical objects. As discussed in Chapter 4, the CENTAUR work is important to this dissertation because of its logic-based semantic specifications.

The Cornell Program Synthesizer [115, 130] provides both editing and execution support. It is often taken as the canonical example of a syntax-directed programming environment. The Synthesizer was originally designed for a single language, PL/CS, but other Synthesizers have also been created. While the system is claimed to be a hybrid between a tree- and a text- editor, the contexts for tree- and text- editing are distinct. Most editing is tree-oriented. The Synthesizer forces a structural view of the program under construction. This approach is useful for the intended application of the Synthesizer: elementary programming instruction.

The Synthesizer Generator [113, 114] is an extension to the original Synthesizer that allows one to create Synthesizers for different languages. Semantic constraints are described to the Generator using attribute grammars. (More will be said about attribute grammars in Chapter 4.) Much of the research done by the Synthesizer project has concerned efficient evaluation of attribute grammars. Recently, the Synthesizer Generator has been used as a basis for the NUPRL [28] mathematics development and proof environment.

COPE [8, 9] is intended to be a tolerant tool for use by novice programmers. COPE is a complete programming environment, providing both program modification and execution. Keywords of the language are used to select constructs during program entry. COPE creates the appropriate template and inserts it into the document. A document can be edited by moving a portion to an edit buffer, changing the text in the buffer, and then reinserting the contents of the buffer into the program. The parser uses a "differential parsing algorithm" which couples error repair with incremental parsing. The language grammar is

factored into several subgrammars in order to ease the error repair process. The underlying parsing algorithm is LL(1). The COPE work is notable for its concern with error repair and error tolerance.

The PSG [13] system, developed at the Technical University of Darmstadt, is a generator for programming environments. The generated environment includes both editing and execution facilities, much like COPE or the Cornell Program Synthesizer. PSG-generated editors allow textual input, but creation of programs seems to be structure-oriented in the tradition of MENTOR or the Cornell Program Synthesizer.

PSG provides advanced language description techniques based on editing program fragments. A fragment is well-formed whenever it can be embedded into a well-formed program. This technique considerably relaxes the consistency constraints found in most other structure-oriented editors. More will be said about contextual-constraint checking in PSG in Section 4.2.3. In the PSG system, the dynamic semantics of languages are described using denotational semantics.

2.2.3 Syntax-Recognizing Editors

Mark Horton's BABEL [58] was an early multilingual syntax-recognizing editor developed at the University of California, Berkeley. Using incremental LR parsing and adopting the Synthesizer's semantic analysis algorithms, Horton showed that a syntax-recognizing editor that accepts text as its primary input is possible. However, BABEL's performance was unacceptably slow, and it did not provide the capabilities required in a front-end. The incremental parsing algorithm required a full parse tree for reparsing, leading to high storage overhead and a complicated structure for users to edit. Using the full parse tree for reparsing also contributed to the performance problem.

SYNED [44] is a syntax-recognizing editor for the C language developed at Bell Laboratories. It is capable of accepting both keyboard (text-oriented) and menu (structure-oriented) input, although one sacrifices certain capabilities if keyboard input is used. The parsing algorithm used by SYNED is a simple variation on LR parsing. While SYNED is a single-language editor, the developers have developed a model of editing similar to that presented in *Pan*.

The SAGA editor [74], developed by Peter Kirslis as part of the Saga Project, is a multilingual syntax-recognizing editor for programs. The primary emphasis of Kirslis' work was to develop effective incremental parsing and error handling methods for an editor in which most interactions followed the text-oriented model of computation. Like BABEL, the SAGA editor maintains a full parse tree as its internal representation.

BABEL, SYNED, the SAGA editor—all three maintained a full parse tree in order to incrementally parse changes to text. While solving the problem of creating hybrid editors, the use of a full parse tree created two new problems. First, the full tree is very large, and contains many extra nodes needed only by the incremental parsing algorithm. Second, the structure presented to the user is based on the structure of the internal (parse) tree, not on the language as understood by users. Thus structure-oriented editing, though possible, becomes improbable. For example, consider the situation when the tree of Figure 2.4 is being edited and the current cursor is at the tree node marked "id_2". Executing the tree command Up to move the cursor to the parent of the current node will move the cursor

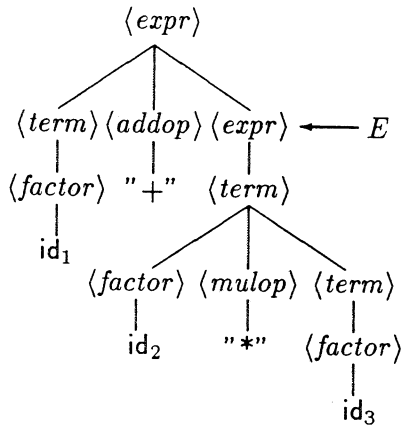


Figure 2.4: Structure Editing in a Simple Parse Tree

to the subtree labeled "⟨*factor*⟩" rather than to the "⟨*expr*⟩" subtree indicated by *E*. No change in the cursor's position will be apparent to the user. Such an approach, although easy to implement, compromises the utility of structure-oriented commands by presenting an incomprehensible structure to the user. Structure editors that use an internal tree based on the abstract syntax of the language, such as MENTOR, can show such aberrant behavior, but generally the abstract syntax corresponds more closely to the user's naive ideas about the underlying structure of the language.

This kind of unpredictable (to the user) structure is not confined to the simple cases shown in Figure 2.4. Any substructure not predictable from simple familiarity with the language leads to undesirable surprises during structure-oriented editing.

There has been a great deal of research in recent years concerning software development environments and their support for programming languages. Among the efforts not previously discussed are those of Alberga *et al.* [6] Reiss [108, 109] and Engels, Nagl, and Schaefer [41]. In general those efforts have addressed issues different from our primary concerns. Consequently their contributions are not summarized here. The proceedings of several recent conferences [27, 54, 55, 56] contain descriptions of many other systems.

2.3 Design Principles for *Pan*

Pan is a multilingual syntax-recognizing editor providing unrestricted text editing. Users of *Pan* can freely intermix text- and structure-oriented operations much like the user of a text-oriented editor can intermix the one- and two-dimensional models of text. *Pan* is extensible and customizable like EMACS. Both keyboard and mouse-oriented interfaces are fully supported.

When designing *Pan*, the desirable characteristics for language-based editors presented in Section 1.1 were distilled into six explicit precepts:

- Integrate text and structure-oriented editing.

- Present the logical structure of documents.
- Support multiple production languages.
- Support the experienced user.
- Create description-driven components.
- Provide a test-bed for user-interface designs.

Additionally, four decisions were determined by the research that *Pan* was intended to support.

- Use incremental parsing to detect syntactic structures.
- Model contextual constraint checking on logic programming.
- Support semantics-driven browsing and presentations.
- Develop a rich model of annotation.

2.3.1 Precepts

The design precepts adopted for *Pan* provided a basis for most major design decisions, particularly with respect to the user interface.

Integrate text- and structure-oriented editing

It is important that users be able to interact with a document either as text or as a structured object. First, documents, especially programs, have both text-like and structural aspects. For instance, a user may replace textually all occurrences of a particular name, both where it is used as an identifier and where it appears in comments. However, the user may also wish to make the change based on the structure of a program, for instance renaming a variable within a certain scope. In the second case, variables having the same name within nested scopes would not change. Second, most kinds of structured documents contain unstructured text, for example, the sentences in a memo or the comments in a program. Third, there are times when a user wants to transform the structure of a document substantially, but wishes to do so efficiently by altering the document's textual representation. For instance, a user may wish to substantially rewrite and reorganize a document, re-using existing pieces that cross structural boundaries. At other times the user may wish to reorganize structurally, for instance by reordering paragraphs or procedure definitions. *Pan* provides both text- and structure-oriented editing for the languages that have been described to *Pan*, yet *Pan* is powerful and efficient enough to be used solely as a text editor.

Present the logical structure of documents

Users should be able to interact with a language-based editor in the terms and concepts of the language being edited. Thus, for a programming language having modules, functions, statements, and expressions, the user should be able to select and operate upon

modules, functions, statements, and expressions. Text editors provide operations only on a stream of characters that may be divided into words and lines. Structure editors, on the other hand, maintain an internal representation of the document's structure within which internal nodes represent the language's substructures. Structure editors, however, do not interpret those nodes, requiring the user to use structure-oriented commands (e.g., Left, Right, In, Out, Delete-Node) rather than more natural language-oriented commands (e.g., Next-Function, Previous-Declaration, Delete-Statement). *Pan* provides language-oriented, rather than structure-oriented, commands, while hiding internal representations from the user. *Pan*'s user interface can be configured to provide only syntax-directed editing[4].

Support multiple languages

Experienced programmers typically use several formal languages in their daily activities—a design or specification language, a structured-documentation language, one or more programming languages. It is essential that a single editing and browsing system support all of these tasks. *Pan* is intrinsically multilingual. Adding a new language consists of writing the description, then generating and loading data tables.

Support the experienced programmer

Experienced programmers require of their tools power, flexibility, and efficiency—three requirements often at odds in the design of a system. Power and efficiency derive, in part, from the data structures and algorithms used to implement the system. Flexibility is determined in part from the algorithms chosen, and in part from the policies enforced by the user interface. For instance, the Cornell Program Synthesizer limits editing power and flexibility by requiring syntax-directed editing [115, 130]. In turn, by assuming syntax-directed editing, the parsing and contextual constraint checking algorithms can be made quite efficient. In designing *Pan*, support for the experienced programmer required that algorithms and data structures be used that can support powerful and flexible operations.

Create description-driven components

Description-driven components separate the description of what a component should do from the specification of how the component should do it. Components such as the parser, the semantic checker, and the tree-presentation modules of *Pan* are description-driven. In general, the descriptions used by components of *Pan* are declarative. Descriptions are converted into language-specific tables that are interpreted by the algorithms of the implementation. For example, LADLE [23] converts the syntax portion of a language description into tables for lexical analysis, parsing, parse tree abstraction, and parse tree expansion.

2.3.2 Research-Driven Design Decisions

Along with the precepts adopted for the design of *Pan*, several specific tactics were established early in the design process. These tactics had to be established early due

[4]Support for top-down (template-driven) tree elaboration would have to be added to use *Pan* effectively as a syntax-directed editor. This topic is discussed in Chapter 3.

to their ramifications throughout the system.

Use incremental parsing to detect syntactic structures

New parsing and tree-building algorithms have been developed for *Pan*. They are based on a newly defined formal relationship between the abstract syntax of a language and its concrete (parsing) syntax called *grammatical abstraction*. Grammatical abstraction allows a decorated abstract syntax tree to be used during incremental LR parsing. Details are given in Chapter 3.

Model contextual constraint checking on logic programming

Contextual constraint checking in *Pan* is modeled on logic programming. The axioms of the semantic definition are established by structure-independent clauses. Goals associated with syntactic structures are used to express and check the context-sensitive constraints of the source language. The information gathered during checking is stored in a logic database that can be accessed by queries written in the logic-based specification language. Creation and use of the database are intertwined; certain conditions must be true before other facts can legitimately be added. The research involved includes the application of logic programming to the description of contextual constraints and the search for efficient approaches to constraint satisfaction in the presence of a changing database. Chapters 4–7 discuss contextual constraint checking in *Pan*.

Provide semantics-driven browsing and presentations

The information gathered by the semantic analyzer can be used by other components of the environment—especially the components used to display and browse a complicated structure. The semantic information is itself a structure that *Pan* will be capable of displaying and browsing. This aspect of *Pan* is part of the research currently being performed by Michael Van De Vanter [132].

Develop a rich model of annotation

Semantic constraints can be regarded as one form of annotation; program commentary is another. *Pan* supports a rich model of annotation using the syntactic and semantic mechanisms already described. Support for linguistic conventions (such as project-specific naming conventions), programming idioms, styles, and standards, for extensible semantics, and for tree-based pattern matching will be provided in order to further extend this model [132].

The research reported here deals with the first two areas of research: parsing and contextual constraint checking. Other aspects of language-based editing—syntax specification [23], display of hierarchical structures [18], methods for annotation, more powerful viewing and browsing mechanisms [132]—are now being addressed by coworkers.

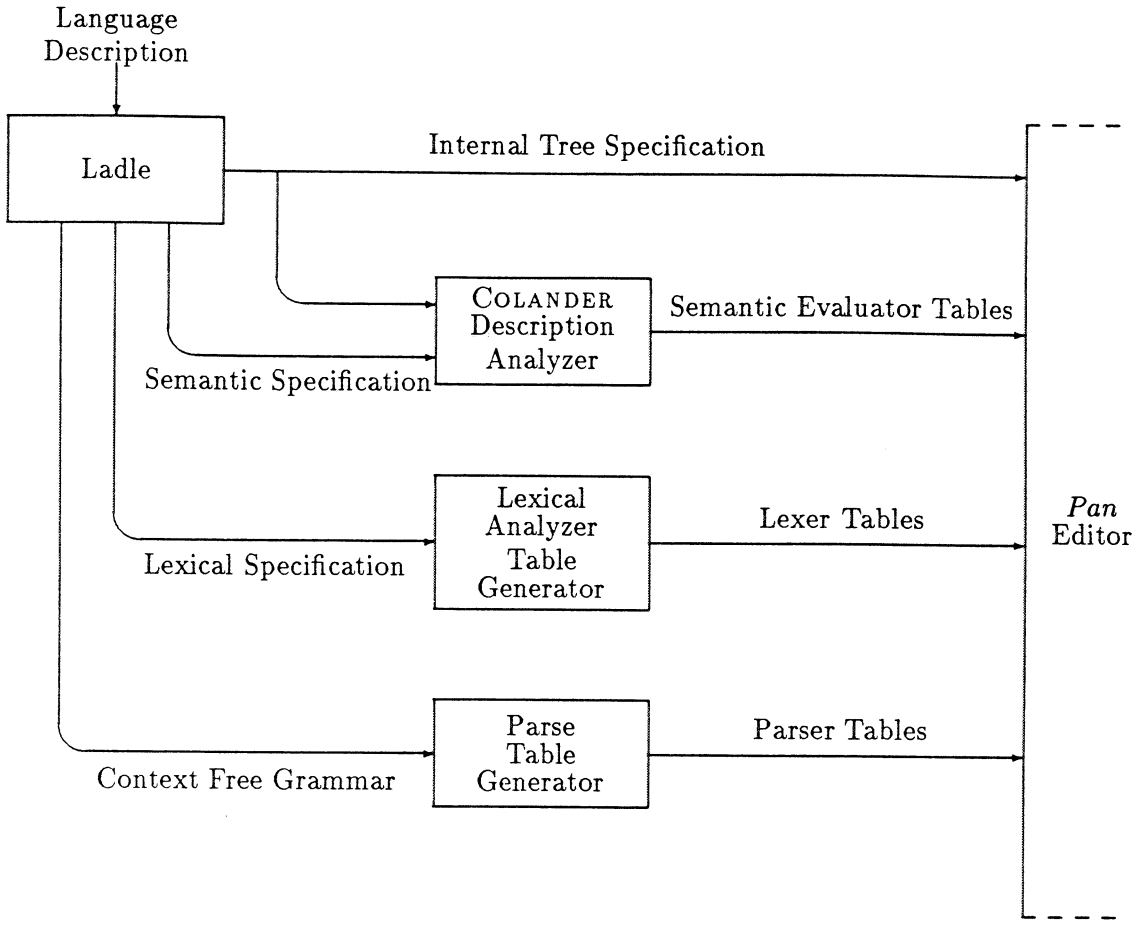


Figure 2.5: Flow of Information among LADLE, COLANDER, and *Pan*

2.4 System Architecture: LADLE, COLANDER, and the *Pan* Editor

Pan is a description-driven editor able to edit many different languages simultaneously[5]. The descriptions used by the *Pan* editor are generated by two preprocessors: the syntax component LADLE [23] and the semantic component COLANDER. Figure 2.5 illustrates the flow of information from a language description through LADLE and COLANDER into *Pan*. Tables can be loaded dynamically during an editor session, although the standard language tables are loaded at editor-creation time. To date, syntactic descriptions have been written for Modula-2, ASPLE, "C" (except preprocessor macros), COLANDER, and for LADLE's own language description language. Semantic descriptions for Modula-2

[5]In the current implementation, each document must be a composed using a single language. The architecture and algorithms support the ability to edit documents composed from multiple languages, but the current implementation does not.

and COLANDER are being developed.

LADLE is a preprocessor for language descriptions. A language description contains four parts that describe the lexical structure, the parsing grammar, the abstract syntax, and the contextual constraints of a language. For structure-oriented editing, only the abstract syntax description is required. If text-oriented editing of syntactic structures is supported, then the information from lexical and syntactic specifications is necessary. The semantic description is required only if semantic checking is to be performed.

LADLE verifies that its input is well-formed by checking various relationships among the four parts. When the description is well-formed, LADLE generates several outputs: input to the lexical analyzer generator, input to the parser generator, and tables describing the language and its internal tree-structured representation for use by *Pan*'s editor commands and by the COLANDER description analyzer. After the tables have been generated, *Pan* loads them to create a language description object.

The lexical analyzer table generator creates tables for use by *Pan*'s lexical analyzers. The input to the lexical analyzer generator is a description of the lexical portion of a language. The description can include regular expressions and expressions denoting bracketed expressions. The bracketing can be either nested or non-nested. The output of the lexical analyzer generator is a collection of tables.

Pan's lexical analyzers synchronize a stream of lexemes with an underlying text stream. A full (non-incremental) lexical analyzer recomputes the entire stream of lexemes from scratch. An incremental lexical analyzer updates only the changed portions of the lexical stream. Additionally, an incremental lexical analyzer creates a summary of changes for use by an incremental parser. In the current implementation, an incremental lexical analyzer capable of creating an entire stream of lexemes is provided.

The parse table generator creates LALR(1) parse tables for use by *Pan*'s parsers. The input to the parser generator consists of a description of a context-free language together with links to the language's lexical specification and error recovery information. Output of the parser generator is a collection of tables.

Pan's incremental parsers maintain the tree-structured representation of the object being edited. A full parser creates a tree from scratch; incremental parsers need only update changed areas of the tree by synchronizing the tree with the lexical stream maintained by the lexical analyzers. Currently, only an incremental LR(1) parser is provided. It is capable of parsing a buffer from scratch just like a full parser.

The COLANDER description analyzer analyzes the language description, and creates the tables used by the consistency manager within *Pan*. During editing, the consistency manager (Chapter 6) monitors changes to a semantic database and uses the COLANDER interpreter to update the database, thereby providing an incremental analyzer.

2.5 Functional Components

The *Pan I* editor is composed of seven functional components. These basic units correspond to the modules of its high-level design: user input/output, command definition and execution, text editing, tree editing, syntax analysis, semantic analysis, and information repository interface. Components cannot operate in isolation, however; they depend on one

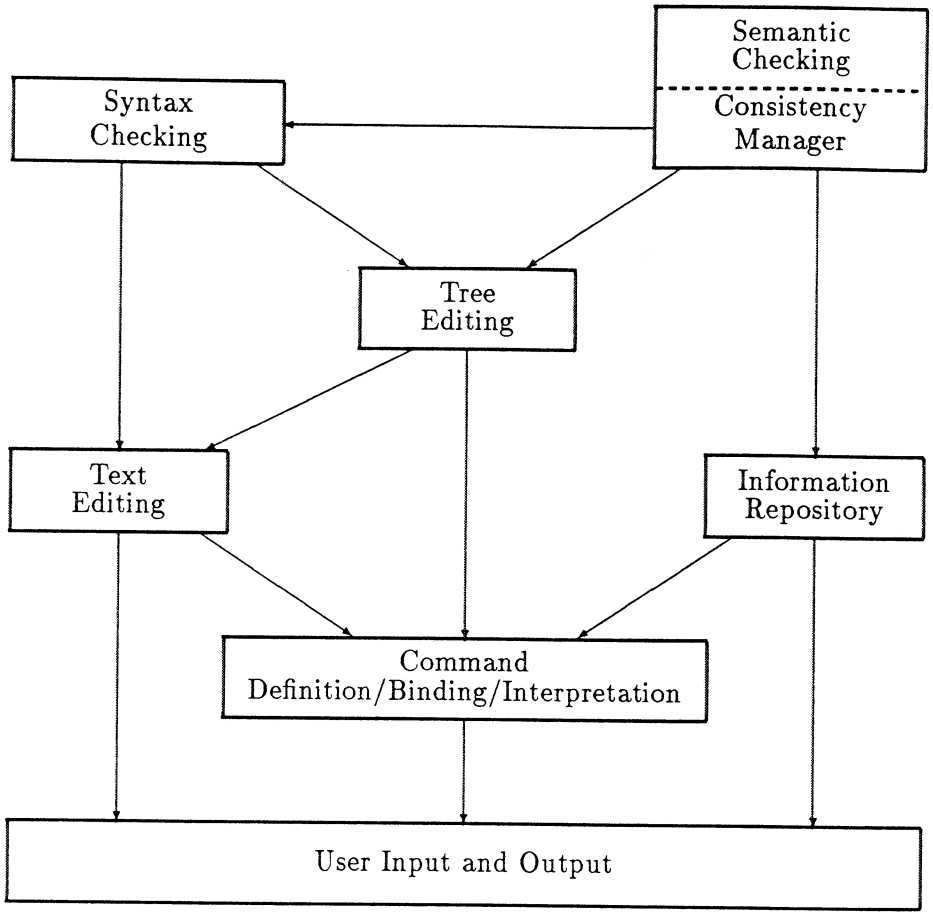


Figure 2.6: Component Dependencies in the *Pan I* Editor

another as shown in Figure 2.6.

A simple editor could be constructed from just three of *Pan*'s functional components: user input and output, command definition and execution, and at least one editable representation (text or tree). More interesting combinations produce more powerful editors as shown in Figure 2.7. The last combination in Figure 2.7 describes is the standard configuration for *Pan*.

The remainder of this section describes *Pan*'s functional components in more detail. Basic functionality is described for each component and subcomponent. Finally, the adequacy of the architecture is discussed.

User I/O + Commands + Text	= Text editor
User I/O + Commands + Tree	=Tree-structure editor
User I/O + Commands + Text + Tree	= Structured document editor
User I/O + Commands + Text + Tree + Syntax	= Syntax-recognizing editor
User I/O + Commands + Text + Tree + Syntax + Semantics + Repository	= Full *Pan*

Figure 2.7: Interesting Combinations of *Pan*'s Functional Components

2.5.1 User Input and Output

The user input and output component is the base level for the entire system. It provides facilities for event handling (user input), screen management, window management, timers, annunciators (user messages), prompters, menus, imaging, and communication with other programs. Most of the user input and output component is implemented in the C programming language. The remainder is implemented in Common Lisp like the other components of *Pan*.

2.5.2 Command Definition and Execution

The command definition and execution component associates user-initiated events with their resulting actions in *Pan*. It has five subcomponents: definition, binding, undo, help, and customization. Many of these components single-handedly provide services to *Pan*'s users.

The command execution loop reads input events, and using the current bindings, maps sequences of events to a chosen editor command. Following execution of the command, the results are passed to the undo facility to be retained for possible later use. This component also provides error signaling and handling facilities for the extension language.

Bindings associate sequences of input events with editor commands. *Pan* provides three forms of bindings, distinguished by how the event sequence originated. Keyboard bindings map a sequence of keyboard events to a command, menu bindings map the selection of a menu item to commands, and operand-level bindings map a pair (operand-level, generic-operation) to a command.

Operand-level bindings generalize the notion of operand modes by supplying an explicit relationship between operand types and operations. In a text editor, commands like Delete-Word combine the operation `delete` with the operand type "word" into a single operation. This combining is reasonable where the set of operand types is both small and well defined.

In a language-based system, the set of operand-types is considerably richer. Along with the basic textual types of character, word, and line, a programming language might define operand types like "lexeme", "syntax error", "declaration", "statement", "expression", or "imperative". A LADLE language description provides a way to group subtree-types in the internal tree representation into operand classes. The set of operand levels for a particular language is called an **operand hierarchy**.

Pan provides a persistent operand-type selector, called the **operand-level**. Each buffer contains an operand type selector that is shared by all viewers open onto that buffer. The current operand-level is persistent: it is changed only by the user. The operand level is most useful in concert with navigation and selection.

Along with the operand-level selector are a collection of generic editing operations: `next`, `previous`, `select`, `mouse-select`, `mouse-extend`, `cut`, `copy`, `paste`, and `delete`.

Operand-level bindings map a pair ⟨`level`,"generic editing operation"⟩ to an editor command. New operand-levels can be added by a user, and current operand level bindings can be changed by a user. Facilities for customizing the operand level bindings include methods for defining new levels, for binding commands at new levels, for changing existing bindings, and for describing the operand level bindings.

Online help facilities provide assistance to the user.

2.5.3 Text Editing

The text component provides a representation of text that can be modified incrementally, display facilities, and primitive editing operations. The text component also supports textual searching and text file input and output.

Various methods are provided for storing and manipulating text objects. In particular, the text representation provides rings (circular bounded stacks) for marks, for deleted text regions (the kill ring), and for text regions shared between different objects (the clipboard).

The text display component maps a textual object onto a display area. This map is currently implemented as an infinite quarter-plane. Horizontal scrolling is supported; line-wrap is not. A display object is called a "viewer" in the *Pan* nomenclature. There may be several viewers sharing a given object. Users always interact with objects through viewers. Each viewer provides a cursor at which operations may occur.

2.5.4 Tree Editing

The tree editing component provides facilities for representing and manipulating n-ary trees by supplying an editable representation for constructing, navigating, selecting, viewing, and editing tree structures. Because the tree-editing component is separate from the syntax-definition component, syntax-directed editors for simple tree structures (such as outlines) can be implemented without using incremental parsing. The syntax-analysis component of *Pan* uses the tree-editing component to represent parsed trees. Specific tree representations are mapped onto the abstraction supplied by this component. Nodes in parsed trees (or semantically checked trees) can be regarded as subtypes of basic tree nodes.

The basic tree representation provides the usual node, parent, and children abstractions. Navigation operations are the standard tree operations (up, down, left-sibling, right-sibling). Selection is by tree cursor—currently implemented as a pointer to a tree node. The only coordinate system provided by the tree component is that of tree node addresses. Tree nodes can be annotated.

2.5.5 Syntax Specification and Checking

The syntax component builds on both text and trees. As a combiner, it uses facilities from both subcomponents to provide editing operations on structures having both text-oriented and tree-oriented aspects. For instance, the "cursor" actually implemented in *Pan* combines a text cursor and a tree cursor. Selecting a subtree also selects the region of text defined by the tree's yield. Deleting a subtree deletes the associated region. The support for editing is implemented by a collection of *Pan* commands.

The syntax checking component provides incremental mappings from text to tree and from tree to text (as necessary). It also provides navigation and selection mechanisms based on the syntax of the language being represented. The syntax component can be divided into several subcomponents: LADLE—the language-definition processor, the lexical table generator, lexical analyzers, the parse table generator, parsers, internal (tree) representation, and support for editing.

2.5.6 Semantics Specification and Checking

COLANDER provides facilities for checking the context-sensitive semantic constraints of a language. It has three subcomponents: a description analyzer, an evaluator, and a consistency manager. The consistency manager (Chapter 6) monitors changes to the syntax tree and the information repository in order to correctly maintain the contextual constraints. Semantics requires the presence of both the tree component and the information repository in order to function. Usually, a semantics component will be used in conjunction with the syntax component. Chapters 4–7 provide more details.

2.5.7 Information Repository Interface

Pan is designed to serve as the front-end for an integrated programming environment. The programs developed, together with the information gathered during the development process, should be stored in a persistent, shared, information repository. Since the appropriate data management system was not immediately available, and since the requirements for a data manager evolved with the *Pan* system, *Pan I* implements its own simple data manager without attempting to provide persistence, locking, and the like. The chapters describing semantic analysis in *Pan* also define the requirements on a program database capable of supporting *Pan*.

2.5.8 Retrospective

The component architecture of Figure 2.6 (page 22) has several shortcomings. In particular, it allows the incremental parser to modify the internal tree directly during parsing. This is efficient for parsing, but it complicates the semantic analysis. Semantic constraints can depend on local tree structure. Since the incremental parser modifies the syntax tree directly, the consistency manager is forced to store duplicate information representing local tree structure in order to detect changes to that structure at a later point. For this reason, the architecture shown in Figure 2.8 would have been a better choice. In Figure 2.8, the dependence of the tree editing component on the text editing component has

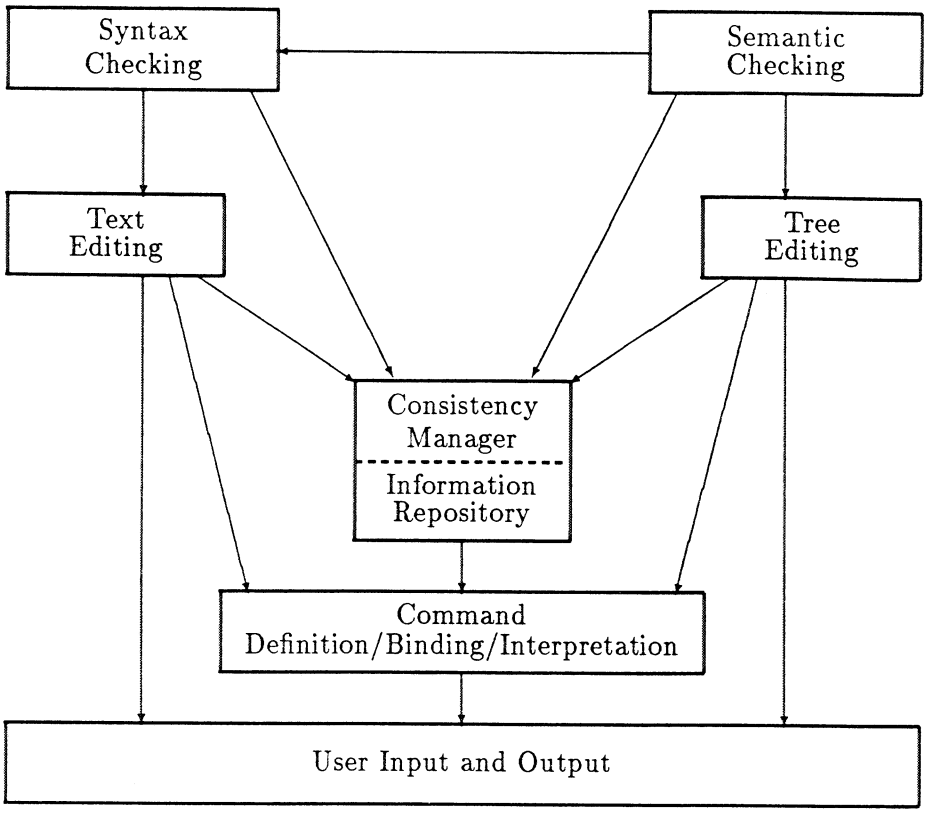


Figure 2.8: Improved Component Architecture for *Pan I*

been removed and the editing components now use the information repository to store their internal structures. In Figure 2.6, consistency management is associated with the semantic checking component. In the proposed model, it is associated with the information repository where it can monitor changes to all shared data structures. The information repository, then, is responsible for notifying other clients of changes to the data under its control. Thus the text, lexeme, and tree structures, as well as the semantic information, should be in the information repository and controlled by the consistency manager.

2.6 Data Representations

The seven functional components of *Pan* provide useful architectural abstractions. However, the components for text and tree editing, syntax and semantic analysis, and the information repository are all highly interrelated. This section, and those that follow, discuss those interrelationships most crucial to understanding the design of *Pan*.

Pan provides data structures for representing both text and tree-structures. In the current implementation, the internal representation of parsed (syntax) trees uses both the primitive text representation and the primitive tree representation. This section dis-

cusses the primitive representations actually in use, their interconnection, and the tradeoffs discovered during their design.

2.6.1 Text

The internal representation of text stores characters in a data structure that can be modified incrementally. The text data structure provides for insertion, deletion, and copying of textual regions, two coordinate systems, and methods for referencing characters. The internal representation of text also provides pointers to the lexical stream (in a parsed-tree representation) that are used to map from characters to lexemes.

Text that is part of the textual representation of a document being edited is called *the text stream*. In the future, it is expected that the text stream will be discontinuous, consisting of isolated pieces of text linked together by other editable representations. Text deleted from the text stream is placed into *deleted text space* from which it can be retrieved.

Text in *Pan* can be conceptualized as either as a stream of characters or as a two-dimensional infinite quarter-plane of characters. Both views are implemented by distinct coordinate systems overlaid on the internal representation.

Each character is a member of a single buffer-relative character syntax class. Character syntax classes are used to implement text cursor motion commands such as searching for balanced brackets or skipping over white space. The character syntax classes defined by *Pan* are: "space", "word character", "punctuation character", "left bracket", "right bracket", and "other".

There are several ways to refer to characters, including character pointers, marks, cursors, and regions. Character pointers refer to a character, not a position between characters. They are implemented using *sticky pointers*, a data structure described by Fischer and Ladner [43]. In *Pan*, where character pointers maintain the mapping between lexemes and underlying text, updating every character pointer on every insertion or deletion would be very costly. The sticky-pointer data structure was chosen because individual sticky-pointers do not have to be updated until they are dereferenced. The drawback to the sticky-pointer implementation is a higher storage overhead.

A **region** is a contiguous sequence of characters. Every command that inserts or deletes text ultimately uses one of the commands Insert-Region or Delete-Region. Commands that operate on single characters, such as the command that deletes a character at the current cursor, act upon implicitly defined regions. The notion of a region is fundamental to editing with *Pan*.

2.6.2 Basic Trees

Every tree node in an internal tree represents an operator in the abstract syntax. Most of the operators have a fixed arity, or number of children. **Sequence nodes** are operators having an arbitrary number of children. This contrasts with systems in which lists of nodes are represented using a binary tree representation.

The advantage to sequence nodes is that they simplify some aspects of the designs of the internal form and the user interface. Their disadvantage is that some aspects of incremental parsing and semantic analysis get more complicated.

2.6.3 On Dual Representations

Pan I provides both text- and structure-oriented editing by maintaining an internal representation of the document that can support either mode of interaction. For simplicity, *Pan I* was designed around a representation in which both the tree-structure and the text stream for the entire document exist simultaneously. Incremental lexical analysis and parsing allow the system to create mappings from the text stream to the parsed tree; a pretty-printing viewer provides the reverse transformation.

While on first sight, the retention of both the text stream and the parsed tree with lexemes appears expensive, it simplifies several aspects of the overall design. First, the internal representation of a parsed document is internally consistent in the sense that editing operations can always move from tree to text and vice versa. Naturally, when moving from text to tree, the text must have been parsed. Parsing is performed automatically in the current configuration. The consistency of the representation contrasts with a scheme in which the parsed (structural) representation is primary, and may contain within it disjoint patches of "demoted" text—text that has been created for editing, but is not yet re-parsed into the structure. In *Pan*, unparsed text resides in the complete text stream where it is automatically available to text-oriented operations.

Second, since one goal of the project was to support full text editing, the use of the text representation as a basis for the parsed representation ensured that text-oriented operations such as scrolling and regular expression searches over text would always be available. Such operations become difficult or inefficient to implement when one converts from the structure to the text as needed. If the system has to reconstruct the text too often, then it becomes advantageous to retain the textual representation in the first place.

Third, retaining the text provided a place to keep track of what text had changed between scans and parses, thus allowing us to experiment with the user interface. It has been a goal of *Pan* to allow arbitrary textual manipulations between parses, and so a simple means for detecting changed and unchanged text was desired. Retaining the text for a document in an unbroken stream thus simplified the incremental scanner.

Finally, in the earliest prototypes, the viewing and imaging components for pure text could be used to display parsed structures as well, allowing us to focus on the more interesting aspects of the research. As *Pan* matured, the close connection between the text stream and the text-oriented imager and the text stream had to be broken in order to support more powerful pretty printers.

The cost of this scheme, of course, is that the full lexical and textual representation must be maintained—at a cost in storage, primarily. Eliminating the text and the lexemes that can be automatically regenerated from the abstract syntax tree saves storage, but increases the complexity of the implementation. Another viable alternative is to eliminate redundant lexemes while preserving the text stream. The latter would save a large amount of storage while preserving the benefits of the dual-representation. The designers of *Pan II* intend to address these problems in the context of more general structures than trees.

Figure 2.9 shows a simplified view of the internal parsed tree data structure. The leaves of the tree, comprised of the lexemes of the language, are linked bidirectionally for easy traversal. Lexemes that are not part of the tree, such as comments, are also linked into the list of lexemes. Each leaf contains a pointer (implemented by a sticky-pointer) to the

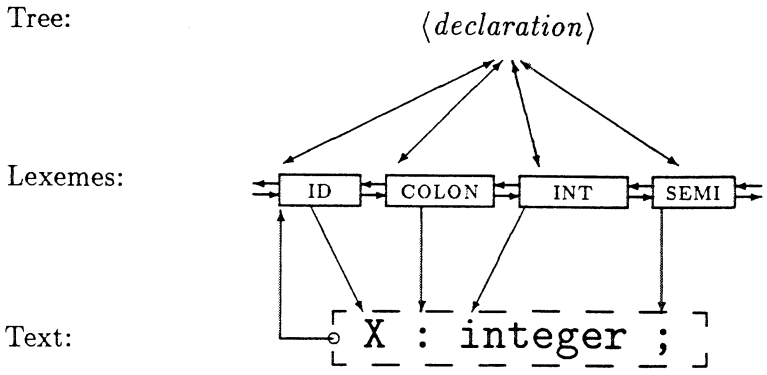


Figure 2.9: Simplified View of Internal Parsed Tree Data Structure

first character of its instance in the text stream. Nodes internal to the tree contain pointers to their children, including their child lexemes. Each tree node also contains a pointer to its parent. An indirect mapping from text to lexemes is also present; each unit of text (up to a full line) has a pointer to the lexeme that contains the first character of text from that unit. This is shown in Figure 2.9 by the pointer from the dashed box back to the ID lexeme.

The stream of lexemes maintained by a lexical analyzer need not be directly editable. It is possible to provide the user with the illusion of traversing and editing the lexical stream using operations defined on the parsed tree representation.

The trees constructed by a parser approximate parse trees. Structurally, they resemble abstract syntax trees decorated with sufficient additional information to enable an incremental parser to use the tree as a base for parsing. Other slots may be provided for semantic information, or for attributes maintained by other tools. Note that the implementation of parsed trees described here is based on design considerations, and not on constraints imposed by our algorithms. In particular, the implementation retains *more* information that our incremental parser requires.

2.7 Scanning, Parsing, and Contextual Constraint Checking

Since *Pan* provides unrestricted text and structure-oriented editing with full checking for syntactic or contextual errors, the system must be able to incrementally update the syntax tree from changes in the text stream. The currently implemented incremental parser uses a full representation of the lexical stream. Synchronizing the syntax tree with the text stream consists of two phases: incremental lexical analysis followed by incremental parsing. Once the syntax tree has been updated, contextual-constraint checking is triggered.

Pan must determine when to attempt rescanning and reparsing. With unrestricted textual operations, it is not simple to determine when a document is likely to be syntactically consistent again. One desirable characteristic of a language-based editor is to provide full and

rapid feedback concerning possible errors in the document being edited. Synchronizing the tree too frequently, (e.g., after every character insertion or deletion) is as detrimental as not synchronizing often enough. First, frequent synchronization leads to spurious error messages: the system detects errors because the user is in the midst of some transformation. At that point, the user presumably knows that the document contains inconsistencies. Second, the performance of the editing system is degraded by over-eager resynchronization.

When to rescan, reparse, and recheck contextual constraints is a policy decision in *Pan*. Many different policies can be supported. Currently, *Pan* implements a "demand-oriented" policy in which rescanning and reparsing is triggered by a demand for structure-oriented information. For example, if the user has made textual changes to the document and then attempts to select a language construct such as a "statement", rescanning and reparsing are triggered automatically. At present, contextual constraints are rechecked immediately after reparsing.

2.7.1 Error Handling

During editing, most documents are incomplete and inconsistent. Error detection and recovery play a crucial role in making *Pan* an effective editor. Errors can occur during any of the three stages of document analysis: lexing, parsing, or constraint checking.

In *Pan*, only the *presence* of errors is signaled to the user. The user is free to examine or to ignore the errors. The special operand levels "syntax error" and "semantic error" can be used to select such errors and to navigate from error to error. When visiting a subtree that represents a syntax error or that has unsatisfied contextual constraints, a message with that error is displayed.

Lexical errors are unrecoverable. (If they were recoverable, then lexical analysis would terminate successfully, but a syntax error would result.) For instance, an unterminated comment may lead to a lexical error. Errors during lexical analysis inhibit both parsing and contextual-constraint checking.

Syntax errors are detected by the incremental parser. At this stage, error recovery is important. In *Pan I*, the error recovery mechanism is based a simple, effective, panic mode mechanism [29]. Directives in the LADLE description for a language are used to tune the error recovery mechanism to that language.

During recovery, the parser gathers up the lexemes and subtrees that are discarded as the recovery proceeds, and makes them children of a special "error" subtree. An error subtree has a variable number of children. The structure and semantic attributions of the subtrees discarded during error recovery are preserved. This error recovery strategy is similar to that used in the SAGA editor [74].

Syntax errors do not delay checking of contextual constraints, so semantic errors can be detected in a syntactically incorrect tree. During contextual constraint checking, the consistency manager ignores any subtrees within an error subtree.

Errors occur during contextual constraint checking when a constraint cannot be satisfied.

2.7.2 Detecting Alterations to the Abstract Syntax Tree

Unlike syntax-directed editors, *Pan* allows great flexibility in the actions a user can undertake. At certain points, *Pan* reparses the altered areas of the document incrementally. The consistency manager underlying the semantic checker depends upon *Pan*'s incremental parser to summarize changes to the internal tree.

Pan's internal syntax tree is modified directly by the incremental parser. Following reparsing, the nodes of the internal tree are classified into four categories relative to parsing: newly created, deleted from tree, possibly changed, and unchanged[6]. The incremental parser is responsible for communicating the status of each node to the consistency manager.

Changes to the internal tree during incremental parsing must be minimized for efficient incremental semantic analysis. The incremental parsing algorithm (see Section 3.5) operates by breaking the internal tree into a forest of subtrees and then using those subtrees to build up a new internal tree. In parsing a change, a simplistic implementation would continually delete and then create tree nodes at the upper levels of the tree on the path from the root to the affected area. If these nodes have semantic data associated with them, the effect on semantic analysis would be extremely costly, since widely-shared data often appears closer to the root of the tree. When the modification is isolated to a single subtree, then the nodes on the path from the root to the modified tree should be preserved with their annotations intact.

During incremental parsing, *Pan* attempts to reuse the syntax tree nodes that were divided during the first phase of parsing. Reuse can be either physical or logical, provided that any annotations or other data associated with the node are preserved.

Node reuse is based on a heuristic that operates well in practice: the parser keeps a stack of tree nodes that were divided; when new nodes are needed they are taken from this stack if appropriate. The heuristic adopted by *Pan* is that a tree node can be reused during parsing if the node will represent the same production in the abstract syntax as in its previous use and its leftmost child is unchanged between parses. Sequence nodes are reused if either the leftmost child or rightmost child is unchanged. Nodes reused during parsing are classified as "possibly changed"; their semantic attributions must be reviewed by the semantic checker.

Incremental constraint checking depends critically on the ability of the incremental parser to reuse nodes in the internal tree. Without this ability, the consistency manager will continually be recreating large amounts of data. With reuse, the semantic values associated with reused nodes are retained across the reuse, even though the values may require updating.

2.8 Interleaving Parsing and Constraint Checking

Most language-based editors, including *Pan*, separate lexical, syntactic, and semantic analysis into distinct phases. None of the editors investigated during this research

[6] A better approach is to embed the internal representation directly in the database maintained by the consistency manager, as discussed in Section 2.5.8. However, the classification of nodes by the parser will still be required.

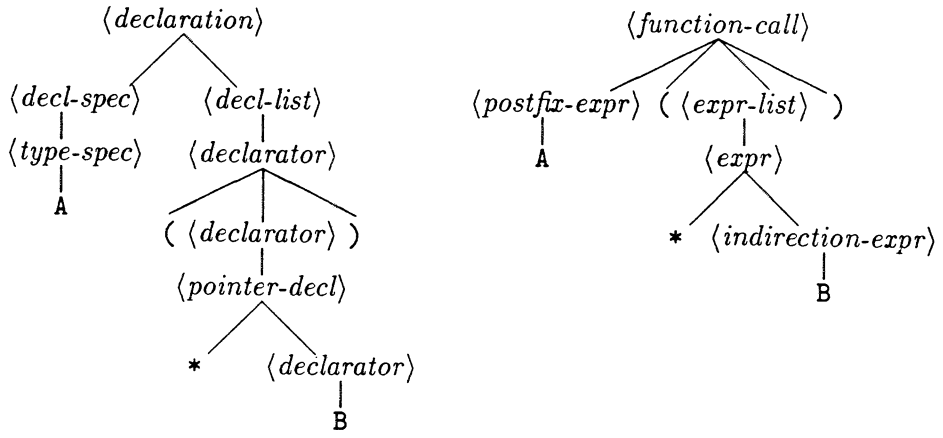


Figure 2.10: Two parse trees for ⟦ A (*B) ⟧ in C

appear able to successfully interleave the phases in full generality. Since these scanning, parsing, and constraint checking are performed sequentially, it is difficult in *Pan* to create parsers or lexers that depend upon semantic information.

An example of parsing that depends on semantic information arises in C. The C language construct ⟦ A (* B) ⟧ is perfectly acceptable C for either a declaration where ⟦ A ⟧ is a type identifier defined by a **typedef**, or for a function call. Figure 2.10 shows two (simplified) parses for this construct: on the left, is a parse tree for a declaration, while on the right is a parse tree for a function call.

In a traditional pass-oriented compiler, the ambiguity is avoided by installing type names into the symbol table as soon as a type definition is recognized during parsing. The lexical analyzer then distinguishes between type identifiers and other identifiers by performing a symbol-table lookup and returning the appropriate lexeme to the parser.

This solution depends on two conditions: (i) that a single pass over the source text assures that the symbol table is consistent and complete and (ii) that the decision whether an identifier is actually a type identifier never needs to be reversed. In a language-based editor, neither condition necessarily holds. While certain assumptions can be used to simplify the problem, the ultimate solution is taxing.

The problem, of course, is to keep the semantic information consistent while processing changes. In *Pan*, the assumption has always been made that the "symbol table" for a program can be created and propagated top-down in the syntax tree[7]. Thus while scanning a lexeme, it is possible that a new scope, containing new or re-defined typedef names, has been initiated in the text, but has not yet been completed. Handling this situation correctly requires an on-the-fly evaluation strategy that makes the actual language description much more complex than is desirable.

The problem is simplified if typedefs nested inside functions or other type definitions are disallowed. In this case, a special "symbol table" for typedef names can be

[7]Only the structure of the symbol table is created this way. Information in the symbol table is added and removed incrementally as discussed in Chapters 4–7.

introduced. This symbol table is augmented when a typedef is recognized during parsing, and is consulted by the lexical analyzer.

Unfortunately, while this solves the problem of parsing correctly, it does not completely handle the consistency-maintenance problem. Consider the code fragment:

```
typedef ... A;                    /* line 1 */
A ( * B );                        /* line 2 */
B++;
```

After parsing, the identifier ⟦ A ⟧ is considered a type identifier. Then line 2 would be considered a declaration for ⟦ B ⟧, and would be added to the declarations for the current block.

But what happens when the typedef is removed or changed, so that ⟦ A ⟧ is no longer a type identifier? Then the original parse for ⟦ A (* B) ⟧ is no longer correct, and the editor must either rescan and reparse the expression or must perform two tree transformations: (i) changing the subtree from declaration to function call (or vice versa), and (ii) moving the transformed subtree from the end of the declarations to the beginning of the statements in the block (or vice versa). Transformation of a declaration into a function call is valid only when the declaration appears at the end of the declarations section; transformation of a function call into a declaration is valid only when the function appears as the first statement in a block. In other cases, errors must be signaled.

Suppose that the ambiguity is resolved during semantic analysis, instead. The same tree transformations would be required, but now one wouldn't attempt to distinguish the constructs during parsing. Instead, the language-description author would have to introduce a special construct for the offending syntax: call it a "*humdinger.*" During semantic processing, the constraints on a humdinger would have to map the construct to either a declaration or a function call as appropriate. Moreover, later failure of the constraint will require the inverse mapping (back to a humdinger-node) to be performed. (COLANDER provides a mechanism that can be adapted to reverse the tree transformations automatically.) The addition of tree transformations is necessary in either approach, while the second approach wins for elegance and correctness.

The drawbacks to this second approach are that (i) the parser will still have reduce-reduce conflicts for humdingers, and (ii) one needs to characterize the possible humdingers correctly.

A third approach is to build knowledge of the alternative parses into the semantic description. For example, once an identifier is found to be a type identifier, the system could reclassify the identifier lexeme by modifying the internal representation of the lexeme and then triggering reparsing to transform the tree correctly. This approach effectively implements the tree transformations by carefully directing and using the parser. From an engineering standpoint, this approach may provide the best solution.

Chapter 3

Grammatical Abstraction and Incremental Parsing

This chapter presents the basic approach to incremental parsing that was developed in the course of this research. LADLE [23] preprocesses the language descriptions used by *Pan*. The algorithms for syntax analysis used in *Pan* and LADLE are based on the theory presented in this chapter. An abridged version of the material presented here appeared as a conference paper [14].

3.1 Introduction

Language-processing tools that parse text often require two different descriptions of a language. The **abstract syntax** defines the structures of the language having independent semantic relevance. The **concrete syntax** defines the structures of the language as seen by a user and as required by a parser. Either can be used to describe an internal (tree-structured) representation of a linguistic object within a language-processing system.

In a syntax-recognizing editor like *Pan*, the user can regard a document as text or as a structured object, intermixing those two views freely. *Pan*'s internal representation of a document must therefore support both local incremental parsing of arbitrarily altered text and structure-oriented editing operations. Until now, editors that provided such flexibility adopted internal representations that corresponded closely to the parse tree of the concrete syntax.

An internal representation that is closer to the abstract syntax is preferable for several reasons:

1. The resulting representation can be chosen to correspond closely to the naive notion of the structure of the language. Matching the external structural view as anticipated by the user with the internal structure of the object as represented by the system simplifies user-interface design while preventing unpleasant surprises.

2. The resulting representation generally contains only semantically relevant structures.

3. For a given object, the resulting representation is usually much smaller than the one that corresponds closely to the parse tree of the concrete syntax.

Providing incremental parsing of textual alterations places constraints on the abstract syntax representation. These constraints would be particularly strong if the parser used the abstract syntax tree directly during parsing. It would then be necessary to tailor the abstract syntax to the chosen parsing technology, placing limitations on the way in which semantic structures could be represented. Support for error recovery, or precedence and associativity enforcement for operators, would require modifications to the abstract syntax that conflict with the original goals of simplicity and semantic relevance.

The approach presented here relates the concrete syntax to the abstract syntax in a way that allows the system to recover sufficient information from the abstract syntax representation to provide incremental parsing using the concrete grammar. This approach retains the advantages of a representation that corresponds closely to the abstract syntax.

Two different descriptions are required: (i) a description of the concrete syntax is needed to parse textual representations of an object and (ii) a description of the abstract syntax is needed to define the internal structured representation. Both of these descriptions are defined using context-free grammars. The context-free grammar that describes the textual representation is called the **concrete grammar**; the grammar that describes the abstract syntax is called the **abstract grammar**. Using separate grammars to describe the concrete syntax and the abstract syntax separates the design of the parser from the design of the internal representation.

Sometimes, a more convenient formalism to specify the concrete and the abstract grammars may be desired. Through the use of grammar transformations, the results presented in this paper extend naturally to regular right-part grammars [80]. The details of this extension are given in the report on LADLE [23].

Although the context-free grammar formalism is useful for describing the correspondence between abstract syntax and concrete syntax, it can be awkward for describing all of the aspects of the abstract syntax representation. For that reason, the language description notation is extended to include additional information. A LADLE language description for the syntax of a language has three parts: a context-free grammar that is used for parsing textual representations, a context-free grammar that serves an intermediate step between the parser grammar and the internal tree representation, and other information about the desired structure of the internal representation. In the following discussion, the grammatical correspondence and the internal representations are described and then the use of those ideas in an implementation is summarized.

3.1.1 Grammatical Abstraction

Grammatical abstraction is a relation between two context-free grammars. When it holds, one grammar is said to be an abstraction of the other. This form of abstraction is structural; it does not use semantic information to identify corresponding structures.

The specification of the concrete syntax, the abstract syntax, and the relationship between the two is declarative—no action routines or special procedures need be provided. Unlike other approaches in which one grammar or the other is derived automatically, the two

descriptions can be modified independently. This approach allows a high degree of control over both the structure of an internal representation and the behavior of the system during parsing. Given the two grammars, it is possible to check whether the abstraction relation holds between them. LADLE includes such a checker.

This form of abstraction is motivated by two requirements. First, it must be possible to construct an internal representation as the corresponding concrete syntax is parsed. Second, it must be possible to map from the abstract syntax back to the concrete syntax, both in order to parse changes incrementally, and in order to display the concrete syntax text to the user. In particular, the abstraction relationship assures that:

1. The abstract syntax represents a less complex version of the concrete syntax, but structures of the abstract syntax correspond to structures of the concrete syntax in a well-defined way. Both grammars describe "almost" the same formal language, subject to the renaming or erasing of symbols.

2. The relationship between the two descriptions is statically verifiable, so that developers can modify either syntax description independently.

3. The transformation from concrete to abstract is reversible, so that relevant information about a concrete derivation can be recovered from its abstract representation. This property allows the system to parse modifications to objects incrementally without preserving the entire derivation relative to the concrete syntax.

4. Efficient incremental transformations from concrete to abstract and from abstract to concrete can be be generated automatically. The transformation from concrete to abstract is triggered by actions of the parser.

3.1.2 Related Work

Transforming a document from an external (concrete syntax) representation to an internal (tree-structured) representation is a well-known problem in language processing. Perhaps the best known solution is the extension of syntax-directed translation called **tree-transduction grammars** [36]. This scheme uses an auxiliary tree-building stack to hold intermediate results during the translation. Each production in a context-free grammar describing the concrete syntax has an associated tree-building action. When a production is recognized, the associated action is performed.

Although the tree transduction approach handles relatively simple cases like flattening chains in expression grammars, it is restricted to being a one-to-one mapping between productions in the grammar for the concrete syntax and productions in the output (transduction) grammar. This restriction imposes undesirable limitations on the author of a language description. First, the author must specify an action explicitly for each production in the concrete grammar, including those with no semantic relevance whatsoever. Second, the description is operational, so the reversibility of the specified transformations is not guaranteed. In order to guarantee reversibility, either the set of available transformations must be constrained, or the reverse transformations must be provided explicitly. Third, the tree-transduction grammars cannot describe some cases in which a limited amount of

local pattern matching (or tree-rewriting) is desirable. The grammars G_1 and $\widehat{G_1}$ of Figure 3.6 (page 42) illustrate one such form of pattern matching where different subtrees are constructed depending on whether the "else clause" is present.

Pattern matching involves relationships among productions in the grammar; it affects the shape of the resulting tree. Syntax-directed schemes fail because patterns necessarily span productions. The notion of a one-to-one mapping between productions in the two grammars is too fine-grained. A complete tree-transformation system, on the other hand, is more general than necessary for incremental parsing. The notion of grammatical abstraction is somewhere in between.

The specification language METAL [67] for MENTOR [38] provides a way to specify local tree-pattern matching. The matching takes place as the internal tree is built. A concrete syntax is described using a context-free grammar. Associated with the concrete grammar is a specification of how to build the internal representation. The specification language permits access to the partially-built tree so that local pattern matching is possible. However, the relationship between the concrete syntax and the abstract syntax is hidden in the procedural specification of the tree-building actions, and is not available to an incremental parser. Since structure-editing is the dominant mode of user interaction supported by MENTOR, no support for fine-grained incremental parsing is provided.

The BETA project [78] also defines abstraction as a relationship between productions in two grammars. The formal definition is based on a mapping from the abstract syntax to the concrete syntax. This mapping is a form of unparsing by which a node in the internal representation is represented textually for display. Unlike MENTOR, there does not seem to be support for bottom-up tree building.

Both Kastens [73] and Rosenkrantz and Hunt [118] relate an abstract syntax to a concrete syntax by deriving a concrete grammar from a grammar describing the abstract syntax. The derivation—specified interactively in Kasten's work, performed automatically in the work of Rosenkrantz and Hunt—is captured by a tool that constructs the transformations to be used during parsing. Noonan [99] takes the opposite approach, deriving a description of an abstract syntax from the grammar used for parsing. His approach is semi-automatic: the developer must intervene in certain cases. Derivation-based approaches ensure the similarity of the two grammars, but do not have the flexibility of modifying the two descriptions independently.

The work of Waddle [134] is close to the work reported here, though developed independently. However, his intent is to reduce the size of the internal tree by eliminating unnecessary nodes. In certain cases, the work reported here will lead to smaller trees.

Finally, the work presented here is strongly related to the notion of grammatical covers [50, 96, 118]. Much of the original work on covers was motivated by the same requirements encountered in this paper.

3.1.3 Examples

The following examples, illustrating several of the above points, are used throughout this chapter. Figure 3.1 summarizes the notational conventions used in this paper. The nonterminal symbol on the left-hand side of the first production of a grammar is taken to be the start symbol. Thus, in Figure 3.2, $\langle expr \rangle$ is the start symbol for G_0, $\widehat{G_0}$, and

Context-Free Grammar:	$G = (V, \Sigma, P, S)$
Terminal symbols:	$a, b, c, \ldots \in \Sigma$ or symbol
Nonterminal symbols:	$A, B, C, \ldots \in N \subseteq V$ or $\langle symbol \rangle$
Arbitrary symbols:	$X, Y, Z, \ldots \in V$
Terminal strings:	$u, v, w, \ldots \in \Sigma^*$
Arbitrary strings:	$\alpha, \beta, \gamma, \ldots \in V^*$
Empty string:	ϵ
Abstract grammar symbols:	$\hat{X}, \hat{\alpha}, \ldots$

Figure 3.1: Notation Pertaining to Context-Free Grammars

$\widehat{G'_0}$. Figure 3.2 shows a simple expression grammar G_0 with two possible abstractions. In the grammar G_0, the nonterminal symbols $\langle factor \rangle$ and $\langle term \rangle$ enforce the precedence of multiplication over addition. The grammars $\widehat{G_0}$ and $\widehat{G'_0}$ provide abstract syntax definitions for G_0 in which precedence and associativity specifications have been removed. In $\widehat{G_0}$, $\langle binop \rangle$ subsumes both $\langle addop \rangle$ and $\langle mulop \rangle$. Parentheses have been eliminated from $\widehat{G_0}$, but all other terminal symbols have been retained. The grammar $\widehat{G'_0}$ provides an alternative abstract grammar for G_0 in which the nonterminal $\langle binop \rangle$ has also been eliminated.

Figure 3.3 illustrates derivation trees for the expression $[\![\mathsf{id}_1 + \mathsf{id}_2 * \mathsf{id}_3]\!]$ using the grammars of Figure 3.2. Figure 3.3.a shows the concrete syntax tree. In the tree of Figure 3.3.b, the reductions '$\langle expr \rangle \rightarrow \langle term \rangle$' and '$\langle term \rangle \rightarrow \langle factor \rangle$' do not appear and the nonterminals $\langle addop \rangle$ and $\langle mulop \rangle$ have been coalesced. The tree in Figure 3.3.c is further compacted. It is easy to see, however, that the subtree rooted at node E in the concrete syntax tree (Figure 3.3.a) corresponds to the subtree rooted at E in the simpler trees. The roots of the trees also correspond. Clearly, the context-free grammar $\widehat{G_0}$ is some sort of abstraction of G_0, and $\widehat{G'_0}$ is some sort of abstraction of both $\widehat{G_0}$ and G_0. The definitions that follow capture this notion.

However, it is the derivation tree of Figure 3.4 that may be desired as an internal representation. In that tree, several extra $\langle expr \rangle$ nodes have been removed. It is possible to define such an abstract syntax by using the context-free grammar of Figure 3.5, but the difficulty of writing a correct specification for the abstract syntax using this style is considerable. An alternative specification method is discussed in Section 3.4.

Figure 3.6 describes the syntax of conditional statements using grammar fragments G_1, $\widehat{G_1}$, and $\widehat{G'_1}$. The two different forms of conditional statement are represented by distinct operators in the internal representation: those having an "else" clause, and those without an "else" clause. While such a distinction may not be required in the actual semantic definition of the language—the existence of an empty statement $[\![\mathsf{null}]\!]$ might allow the semantic equivalence

$$[\![\mathsf{if}\ \langle expr \rangle\ \mathsf{then}\ \langle stmts \rangle]\!] \equiv [\![\mathsf{if}\ \langle expr \rangle\ \mathsf{then}\ \langle stmts \rangle\ \mathsf{else\ null}]\!]$$

to be established—such a distinction is usually desired for readability, formatting, or mapping back to the source language text.

Concrete Grammar G_0

1)	⟨*expr*⟩	→	⟨*term*⟩ ⟨*addop*⟩ ⟨*expr*⟩
2)		|	⟨*term*⟩
3)	⟨*term*⟩	→	⟨*factor*⟩ ⟨*mulop*⟩ ⟨*term*⟩
4)		|	⟨*factor*⟩
5)	⟨*factor*⟩	→	id
6)		|	"(" ⟨*expr*⟩ ")"
7)	⟨*addop*⟩	→	"+"
8)	⟨*mulop*⟩	→	"*"

Abstract Grammar $\widehat{G_0}$

1′)	⟨*expr*⟩	→	⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
2′)		|	id
3′)	⟨*binop*⟩	→	"+"
4′)		|	"*"

Abstract Grammar $\widehat{G'_0}$

1″)	⟨*expr*⟩	→	⟨*expr*⟩ "+" ⟨*expr*⟩
2″)		|	⟨*expr*⟩ "*" ⟨*expr*⟩
3″)		|	id

Figure 3.2: Simple Expression Grammar with Two Possible Abstractions

3.2 Definitions

Suppose $G = (V, \Sigma, P, S)$ and $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ are two reduced context-free grammars, and that $\widehat{G}$ is to be an abstraction of G. Grammatical abstraction relates a subset of the productions in a concrete grammar to each production in the abstract grammar. This is similar to the notion of grammatical covers [50, 61, 96, 118], but the starting point differs.

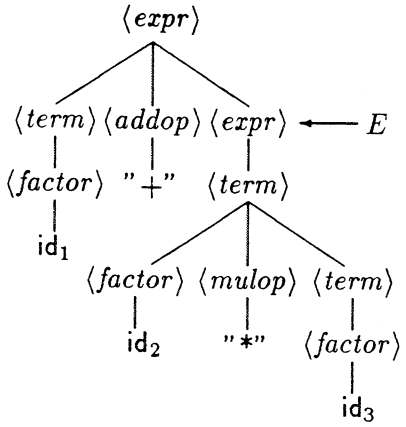
Motivating the definition of abstraction are four ideas:

1. There is a correspondence between a set of *significant* nonterminal symbols in G and the nonterminal symbols in $\widehat{G}$. Intuitively, if the significant symbol $A \in N$ corresponds to the nonterminal $\widehat{A} \in \widehat{N}$, then the two symbols represent corresponding structures in the two languages $L(G)$ and $L(\widehat{G})$. Nonterminal symbols are significant independent of context.

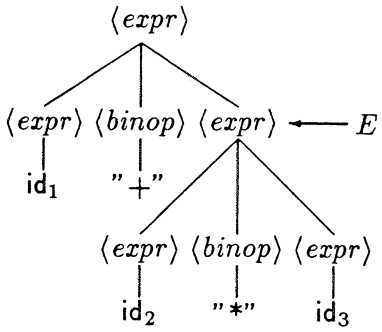
2. Every terminal symbol in G either corresponds to a terminal symbol in $\widehat{G}$ or is absent from $\widehat{G}$.

3. Every derivation in G has a corresponding derivation in $\widehat{G}$. Certain derivations from significant symbols in G are directly related to individual productions in $\widehat{G}$.

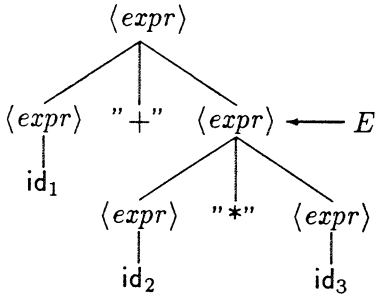
4. Each production in $\widehat{G}$ corresponds to a derivation in G beginning with a significant nonterminal symbol. That is, if the production $\widehat{A} \rightarrow \widehat{\alpha}$ is in $\widehat{P}$ then $A \stackrel{*}{\Rightarrow} \alpha$ in G where the significant nonterminal A corresponds to $\widehat{A}$ and the string α corresponds to $\widehat{\alpha}$.



3.3.a: For concrete grammar G_0



3.3.b: For abstract grammar $\widehat{G_0}$



3.3.c: For abstract grammar $\widehat{G'_0}$

Figure 3.3: Syntax Trees for $[\![\mathsf{id}_1 + \mathsf{id}_2 * \mathsf{id}_3]\!]$

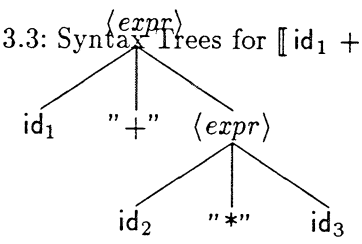


Figure 3.4: Desired Syntax Tree

1)	⟨*expr*⟩	→	⟨*expr*⟩ "+" ⟨*expr*⟩
1.1)		|	id "+" ⟨*expr*⟩
1.2)		|	⟨*expr*⟩ "+" id
1.3)		|	id "+" id
2)		|	⟨*expr*⟩ "*" ⟨*expr*⟩
2.1)		|	id "*" ⟨*expr*⟩
2.2)		|	⟨*expr*⟩ "*" id
2.3)		|	id "*" id

Figure 3.5: Grammar For Desired Internal Tree

Concrete Grammar G_1

1)	⟨*stmt*⟩	→	⟨*if-stmt*⟩ ";"
2)	⟨*if-stmt*⟩	→	⟨*if-part*⟩ ⟨*else-part*⟩
3)	⟨*if-part*⟩	→	if ⟨*expr*⟩ then ⟨*stmts*⟩
4)	⟨*else-part*⟩	→	ϵ
5)		|	else ⟨*stmts*⟩

Abstract Grammar $\widehat{G_1}$

1′)	⟨*stmt*⟩	→	if ⟨*expr*⟩ then ⟨*stmts*⟩ ";"
2′)		|	if ⟨*expr*⟩ then ⟨*stmts*⟩ else ⟨*stmts*⟩ ";"

Abstract Grammar $\widehat{G'_1}$

1″)	⟨*stmt*⟩	→	⟨*expr*⟩ ⟨*stmts*⟩
2″)		|	⟨*expr*⟩ ⟨*stmts*⟩ ⟨*stmts*⟩

Figure 3.6: Conditional Statement Grammars

The definition of grammatical abstraction is based upon a set of **interesting** nonterminal symbols $H \subseteq N$, together with a function θ_0 mapping symbols in $\Sigma \cup H$ to symbols in $\widehat{\Sigma} \cup \widehat{N} \cup \{\epsilon\}$ provided by the language designer. The set of interesting nonterminals H is a shorthand description for the structures of G that are retained in $\widehat{G}$. Depending on the actual grammar G, the set of interesting nonterminals must be extended to include other nonterminals that represent the same abstract structures as those described by nonterminals in H. The resulting extended set of symbols is termed **significant**.

From the set of interesting nonterminal symbols H and the initial mapping θ_0, the set of significant nonterminal symbols and the extended mapping function θ are obtained. Varying H and θ_0 leads to different abstractions of the same concrete grammar.

Unlike covers, the extended mapping function θ is allowed to erase as well as rename symbols. The mapping θ_0 is required to be a function such that $\theta_0(\Sigma) \subseteq \widehat{\Sigma} \cup \{\epsilon\}$ and $\theta_0(H) \subseteq \widehat{N}$. In this case, θ_0 is an **admissible abstraction map.**

Definition 1 Let $G = (V, \Sigma, P, S)$ and $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ be two reduced context-free grammars. Let $H \subseteq N$ be the set of interesting nonterminals of G. Then the mapping $\theta_0 \colon \Sigma \cup H \to \widehat{V} \cup \{\epsilon\}$ is an **admissible abstraction map** if

1. θ_0 is a function.
2. Every terminal symbol in G either has a single corresponding terminal symbol in $\widehat{G}$ or is erased; that is, $\theta_0(\Sigma) \subseteq \widehat{\Sigma} \cup \{\epsilon\}$, and
3. Every interesting nonterminal symbol in G has a single corresponding nonterminal symbol in $\widehat{G}$; that is, $\theta_0(H) \subseteq \widehat{N}$.

The mapping θ_0 can be extended homomorphically to $\theta_0 \colon (\Sigma \cup H)^* \to (\widehat{V} \cup \{\epsilon\})^*$ by

$$\theta_0(\alpha\beta) = \theta_0(\alpha)\theta_0(\beta) \quad \text{and} \quad \theta_0(\epsilon) = \epsilon.$$

∎

The requirement that θ_0 be a function independent of context can be relaxed, in the case of LR(k) grammars, by using a construction similar to one given by Graham [49].

The circularities arising from precedence enforcement are typical sources of *identified* symbols. For instance, the symbols ⟨*factor*⟩ and ⟨*term*⟩ of the expression grammar G_0 are identified with ⟨*expr*⟩ whenever parentheses are eliminated from the abstract syntax. The identified symbols, together with the interesting symbols constitute the set of significant symbols.

Definition 2 Assume that $G = (V, \Sigma, P, S)$ and $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ are reduced context-free grammars. Let $H \subseteq N$ be the set of interesting nonterminals of G, and let $\theta_0 : \Sigma \cup H \rightarrow \widehat{V} \cup \{\epsilon\}$ be an admissible abstraction map. Let $P_{N-H} = \{ A \rightarrow \alpha \mid A \in N - H \}$. Then *Identified* $= \{ B \in N - H \mid \exists A \in H$ such that

$$A \Rightarrow \alpha \underset{P_{N-H}}{\overset{*}{\Rightarrow}} x_1 B x_2 \text{ where } \theta_0(x_1 x_2) = \epsilon \text{ and}$$

$$B \underset{P_{N-H}}{\overset{*}{\Rightarrow}} y_1 A y_2 \text{ where } \theta_0(y_1 y_2) = \epsilon \}$$

When the nonterminal B is added to the set *Identified* because it is reachable from the interesting nonterminal A under the conditions given above, the nonterminal B is **identified** with A. ∎

Note that $H \cap$ *Identified* $= \emptyset$ and that all *identified* symbols participate in recursion in G.

Definition 3 Assume that $G = (V, \Sigma, P, S)$ and $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ are reduced context-free grammars. Let $H \subseteq N$ be the set of interesting nonterminals of G, and let $\theta_0 : \Sigma \cup H \rightarrow \widehat{V} \cup \{\epsilon\}$ be an admissible abstraction map. Let the set *Identified* be the set defined in Definition 2.

The **significant** symbols are defined by *Significant* $= H \cup$ *Identified*.

The remaining **nondenotative** nonterminal symbols reflect steps in a concrete derivation that are elided in the corresponding abstract derivation: *Nondenotative* $= N -$ *Significant*. ∎

In the example grammar G_1, if ⟨*stmt*⟩ $\in H$ and ⟨*if-stmt*⟩ $\notin H$, then for most choices of θ_0, ⟨*if-stmt*⟩ is a nondenotative symbol.

Using the definitions of identified and nondenotative symbols, the initial mapping function θ_0 can be extended to the mapping θ on $\Sigma \cup N$.

Definition 4 Assume that $G = (V, \Sigma, P, S)$ and $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ are reduced context-free grammars. Let $H \subseteq N$ be the set of interesting nonterminals of G, and let $\theta_0 : \Sigma \cup H \rightarrow \widehat{V} \cup \{\epsilon\}$ be an admissible abstraction map. Let *Identified*, *Significant*, and *Nondenotative* be as previously defined. Let "$\perp$" be any symbol not appearing in $\widehat{V}$. (The symbol "$\perp$" is used as a notational device to avoid certain sentential forms of G when considering the correspondence with $\widehat{G}$.)

The following constraint is placed upon θ_0: if A and B are two different interesting symbols, and the nonterminal C is identified with both A and B, then A and B must be treated identically in the abstract syntax. Formally, if $A, B \in H, A \neq B$; C is identified

with A, and C is identified with B, then $\theta_0(A) = \theta_0(B)$. Were it not for this restriction, extra mechanism would be required to map from the abstract syntax back to the concrete syntax.

The extended mapping function $\theta: V \rightarrow \widehat{V} \cup \{ \epsilon, \perp \}$ is defined by

$$\theta(X) = \begin{cases} \theta_0(X) & \text{if } X \in H \cup \Sigma \\ \theta_0(A) & \text{if } X \in \textit{Identified} \text{ and} \\ & \quad X \text{ is identified with } A \\ \perp & \text{if } X \in \textit{Nondenotative} \end{cases}$$

and extend θ to a homomorphism on V^* in the usual way. ∎

Finally, define the set of **significant productions** to be $P_{sig} = \{ A \rightarrow \alpha \mid A \in \textit{Significant} \}$ and let $P_{non} = P - P_{sig}$.

3.3 Weak Grammatical Abstraction

The structures defined from H and θ_0 are used to formalize the notions of weak and strong grammatical abstraction. The definitions of both weak and strong abstraction require that the concrete syntax be described by an unambiguous context-free grammar. This requirement is not overly restrictive, since the concrete syntax will be normally be used to generate parsers. It is possible to generalize the formal treatment to context-free grammars such as those handled by *yacc* [5] in which operator-precedence directives to the parser generator substitute for structures in the context-free grammar. However, that generalization complicates the discussion without providing additional insight to the reader, so it is not included here.

Definition 5 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar with $H \subseteq N$. Let $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ be a reduced context-free grammar. Let *Significant* be as defined previously. Let θ_0 be an admissible abstraction map, with θ the extended mapping function. Then $\widehat{G}$ is a **weak $\langle H, \theta_0 \rangle$-abstraction** of G if

1. $S \in H$ and $\theta(S) = \widehat{S}$, and
2. For every $A \in \textit{Significant}$, and for every derivation in G of the form $A \Rightarrow \beta \overset{*}{\underset{P_{non}}{\Rightarrow}} \alpha$ where $\alpha \in (\Sigma \cup \textit{Significant})^*$, either $\theta(A) = \theta(\alpha)$, or the production $\theta(A) \rightarrow \theta(\alpha)$ appears in $\widehat{P}$.

∎

The definition of weak abstraction allows nondenotative symbols to be recursive. However, it is the case that when $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G, if $A \in \textit{Nondenotative}$ and $A \overset{*}{\underset{P_{non}}{\Rightarrow}} x_1 A x_2$ where $x_1, x_2 \in (\Sigma \cup \textit{Significant})^*$, then $\theta(x_1 x_2) = \epsilon$. This means that circularities involving only nondenotative symbols cannot add any symbols that are represented in the abstract syntax.

Consider again the grammars G_0 and $\widehat{G_0}$ of Figure 3.2. The grammar $\widehat{G_0}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G_0 when $H = \{ \langle expr \rangle, \langle addop \rangle, \langle mulop \rangle \}$ and the initial mapping θ_0 is given by

$$\theta_0(X) = \begin{cases} X & \text{if } X \in \{ \langle expr \rangle, \mathsf{id}, \text{"+"}, \text{"*"} \} \\ \langle binop \rangle & \text{if } X \in \{ \langle addop \rangle, \langle mulop \rangle \} \\ \epsilon & \text{if } X \in \{ \text{"("}, \text{")"} \}. \end{cases}$$

In this case, *Identified* $= \{ \langle term \rangle, \langle factor \rangle \}$ and *Nondenotative* $= \emptyset$. The extended mapping θ is given by

$$\theta(X) = \begin{cases} X & \text{if } X \in \{ \mathsf{id}, \text{"+"}, \text{"*"} \} \\ \langle binop \rangle & \text{if } X \in \{ \langle addop \rangle, \langle mulop \rangle \} \\ \langle expr \rangle & \text{if } X \in \{ \langle expr \rangle, \langle term \rangle, \langle factor \rangle \} \\ \epsilon & \text{if } X \in \{ \text{"("}, \text{")"} \}. \end{cases}$$

If the parentheses were not erased, the grammar $\widehat{G_0}$ would *not* be a weak abstraction of G_0 given the above definition of H.

The grammar $\widehat{G'_0}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G_0 when $H = \{ \langle expr \rangle \}$ and θ_0 is given by

$$\theta_0(X) = \begin{cases} X & \text{if } X \in \{ \langle expr \rangle, \mathsf{id}, \text{"+"}, \text{"*"} \} \\ \epsilon & \text{if } X \in \{ \text{"("}, \text{")"} \}. \end{cases}$$

The nonterminal symbols $\langle addop \rangle$ and $\langle mulop \rangle$ are nondenotative symbols in this example.

Both of the grammars $\widehat{G_1}$ and $\widehat{G'_1}$ in Figure 3.6 are weak abstractions of G_1 when $H = \{ \langle stmt \rangle \}$. For $\widehat{G_1}$, all terminals symbols shown are preserved, while in $\widehat{G'_1}$, the terminal symbols other than those embedded in $\langle expr \rangle$ or $\langle stmts \rangle$ are erased. (The example does not describe the structure of $\langle expr \rangle$ or $\langle stmts \rangle$.)

Theorem 1 shows that weak grammatical abstraction preserves the language and its structure up to the differences induced by the mapping θ. That is, if $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G then $\theta(L(G)) \subseteq L(\widehat{G})$. Furthermore, every sentence in $L(G)$ has a structurally similar representative in $L(\widehat{G})$.

Theorem 1 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar, and let $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ be a reduced context-free grammar. Let *Significant* be as defined previously and let θ be the extended mapping function of θ_0.

If $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G, then for all $A \in H$, if $A \overset{*}{\underset{G}{\Rightarrow}} \alpha$ where $\alpha \in (\Sigma \cup \mathit{Significant})^*$ then $\theta(A) \overset{*}{\underset{\widehat{G}}{\Rightarrow}} \theta(\alpha)$.

Proof Let $A \in H$ and $A \overset{*}{\underset{G}{\Rightarrow}} \alpha$ where $\alpha \in (\Sigma \cup \mathit{Significant})^*$. The proof is by induction on the number n of productions in P_{sig} used in the derivation of α.

Base: $n = 0$. Since $A \in H \subseteq \mathit{Significant}$, this case can only occur when no productions at all are used. The definition of θ (Definition 4) insures that $\theta(A) \in \widehat{N}$ in this case. Since $\alpha = A$, $\theta(A) \overset{*}{\underset{\widehat{G}}{\Rightarrow}} \theta(\alpha)$ trivially.

Inductive Argument: Now assume that the theorem is true for all derivations using fewer than n productions in P_{sig}. Suppose $A \overset{*}{\Rightarrow} \alpha$ where $\alpha \in (\Sigma \cup \mathit{Significant})^*$ using $n + 1$ productions from P_{sig}. Choose any one such derivation.

Then for some $C \rightarrow \gamma \in P_{sig}$ which is the $n+1$st production in P_{sig} used in that derivation, and for some $\alpha_1, \alpha_2, \alpha_3 \in (\Sigma \cup \mathit{Significant})^*$ such that $\alpha = \alpha_1\alpha_2\alpha_3$,

$$A \stackrel{*}{\Rightarrow} \alpha_1 C \alpha_3 \Rightarrow \alpha_1 \gamma \alpha_3 \underset{P_{non}}{\stackrel{*}{\Rightarrow}} \alpha_1\alpha_2\alpha_3 = \alpha$$

By the induction hypothesis $\theta(A) \stackrel{*}{\Rightarrow} \theta(\alpha_1 C \alpha_3) = \theta(\alpha_1)\theta(C)\theta(\alpha_3)$.

Since $C \Rightarrow \gamma \underset{P_{non}}{\stackrel{*}{\Rightarrow}} \alpha_2$, the definition of weak abstraction assures us that either $\theta(C) = \theta(\alpha_2)$ or the production $\widehat{C} \rightarrow \widehat{\alpha_2} \in \widehat{P}$. In either case, $\theta(C) \stackrel{*}{\Rightarrow} \theta(\alpha_2)$.

Hence $\theta(A) \stackrel{*}{\Rightarrow} \theta(\alpha_1)\theta(C)\theta(\alpha_3) \Rightarrow \theta(\alpha_1)\theta(\alpha_2)\theta(\alpha_3) = \theta(\alpha)$.

Corollary 1 If $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G, then $\theta(L(G)) \subseteq L(\widehat{G})$.
Proof Follows from Theorem 1 and the fact that $S \in H$ and $\theta(S) = \widehat{S}$.

3.4 Building the Internal Representation

Suppose that $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G where $\widehat{G}$ describes the abstract syntax and G describes the concrete syntax of a language. The definition of weak $\langle H, \theta_0 \rangle$-abstraction establishes a set of pairs $\langle A \Rightarrow \beta \underset{P_{non}}{\stackrel{*}{\Rightarrow}} \alpha, \widehat{A} \rightarrow \widehat{\alpha} \rangle$ that associate derivations in G with productions in $\widehat{G}$. The task of the tree-building component of a language-processing system is to recognize when the derivations in G are sufficiently complete that the subtree representing that derivation can be constructed. The tree-builder also implements the mapping θ.

Using the abstract grammar to define an internal representation has limitations in practice. The abstract grammar may include chain productions that need not be represented structurally in the internal representation. The author of a LADLE language description can indicate whether a chain production in the abstract grammar should be represented structurally, by an annotation on an existing subtree, or completely omitted. This treatment of chain productions is simply an encoding of singleton subtrees. The mapping θ may induce certain chain derivations in the abstract syntax. This issue is discussed in Section 3.5.

During bottom-up parsing of the concrete syntax, a derivation corresponding to a production in the abstract grammar is always completed by the reduction of a production in P_{sig}. (For this reason, productions in P_{sig} are called *significant productions*.) At this point in the parse, a new node in the internal representation can be created. Building the internal representation during bottom-up parsing requires an additional stack to hold subtrees until they are integrated into larger trees. When a tree node is to be created, its subtrees will be found on the tree-building stack. This stack may not parallel the parse stack, since intermediate reductions can change the parse stack without changing the tree-building stack.

There might exist two forms $\alpha_1, \alpha_2 \in (\Sigma \cup \mathit{Significant})^*$ such that $\theta(\alpha_1) \neq \theta(\alpha_2)$ and both $A \Rightarrow \beta \underset{P_{non}}{\stackrel{*}{\Rightarrow}} \alpha_1$ and $A \Rightarrow \beta \underset{P_{non}}{\stackrel{*}{\Rightarrow}} \alpha_2$ are induced by weak $\langle H, \theta_0 \rangle$-equivalence. In

Concrete	Abstract
3	$3', 4'$
4	$5', 6', 7', 8'$
5	$9', 10'$

Table 3.1: Correspondences between Concrete and Abstract Productions

this case, the two different derivations must be represented distinctly in the internal representation since they represent different semantic constructs. The reduction of the production $A \rightarrow \beta$ will not provide enough information to construct the tree properly.

For every element of P_{sig} mapping to more than a single production in the abstract grammar, the system must be able to distinguish during parsing which production in the abstract grammar corresponds to the production in P_{sig} being reduced.

Two different methods have been investigated for selecting the correct production in the abstract grammar to use when a significant production is being reduced. The first approach modifies the concrete grammar so that each production in P_{sig} corresponds to a single production in the abstract grammar. This approach can be seen as a way to merge the choice of abstract productions into the state machine of the parser. The second approach moves the mechanism out of the parser by encapsulating derivation sequences using values, called *yield-states*, that are assigned to reductions in the parse. During a reduction action, a new yield-state is computed from the states of the right-hand side elements on the parse stack. In effect, the second approach introduces a separate state machine for the abstraction process.

Figure 3.7 contains several examples of this situation. (For simplicity, the structure of expressions is not shown.) Assume that $H = \{ \langle stmts \rangle, \langle stmt \rangle, \langle expr \rangle \}$, and $\theta_0(X) = X$ for all elements of $H$and Σ. Then *Significant* $= H$ and P_{sig} consists of the productions numbered 1, 2, 3, 4, 5, 6 and 15. Some of the correspondences between productions in the concrete grammar and those in the abstract grammars of Figure 3.7 are shown in Table 3.1. In each case, when the reduction of a rule in the concrete grammar occurs, the tree-building action corresponding to the appropriate production in the abstract syntax must be invoked. A common situation arises from the "if" statement as in productions 3, 7, 8, and 9.

3.4.1 Grammar Modification

Grammar modification alters the concrete grammar so that a production in P_{sig} relates to only one production in the abstract grammar. Modifying the grammar is one way to split states within the LR parser. It does not require any knowledge of the derivation sequences encountered, and can be performed either manually or automatically.

Grammar modification proceeds by finding the branches in the derivation and propagating them "backward" to the initial production. By renaming the alternative productions, duplicating productions, and propagating the differences toward the initial production, one can move the alternatives to the "top" of the derivation. This results in some number of new significant productions, each one corresponding to a single production in $\widehat{P}$.

Concrete Grammar

1)	⟨*stmts*⟩	→	ϵ
2)		|	⟨*stmts*⟩ ⟨*stmt*⟩
3)	⟨*stmt*⟩	→	⟨*if-stmt*⟩
4)		|	⟨*loop*⟩
5)		|	⟨*label*⟩ begin ⟨*stmts*⟩ end
6)		|	exit ⟨*label*⟩
7)	⟨*if-stmt*⟩	→	if ⟨*expr*⟩ then ⟨*stmts*⟩ ⟨*else-part*⟩
8)	⟨*else-part*⟩	→	else ⟨*stmts*⟩
9)		|	ϵ
10)	⟨*loop*⟩	→	⟨*label*⟩ while ⟨*expr*⟩ ⟨*stmts*⟩ ⟨*loop-tail*⟩
11)	⟨*loop-tail*⟩	→	repeat
12)		|	unless ⟨*expr*⟩
13)	⟨*label*⟩	→	ϵ
14)		|	id
15)	⟨*expr*⟩	→	...

Abstract Grammar

1′)	⟨*stmts*⟩	→	ϵ
2′)		|	⟨*stmts*⟩ ⟨*stmt*⟩
3′)	⟨*stmt*⟩	→	if ⟨*expr*⟩ then ⟨*stmts*⟩
4′)		|	if ⟨*expr*⟩ then ⟨*stmts*⟩ else ⟨*stmts*⟩
5′)		|	while ⟨*expr*⟩ ⟨*stmts*⟩ repeat
6′)		|	while ⟨*expr*⟩ ⟨*stmts*⟩ unless ⟨*expr*⟩
7′)		|	id while ⟨*expr*⟩ ⟨*stmts*⟩ repeat
8′)		|	id while ⟨*expr*⟩ ⟨*stmts*⟩ unless ⟨*expr*⟩
9′)		|	begin ⟨*stmts*⟩ end
10′)		|	id begin ⟨*stmts*⟩ end
11′)		|	exit
12′)		|	exit id
13′)	⟨*expr*⟩	→	...

Figure 3.7: Grammars Showing Alternative Productions in the Abstract Grammar

In the example of Figure 3.7, ⟨*label*⟩ and ⟨*loop-tail*⟩ are non-recursive, nondenotative symbols, each of which has two alternatives. The grammar modification algorithm replaces those alternatives with new nonterminals, substituting { ⟨*namedL*⟩, ⟨*anonL*⟩ } for ⟨*label*⟩ and replacing ⟨*loop-tail*⟩ by { ⟨*unless*⟩, ⟨*repeat*⟩ }. Substitution into production 10 then yields the new productions

10.1) ⟨*loop*⟩ → ⟨*anonL*⟩ **while** ⟨*expr*⟩ ⟨*stmts*⟩ ⟨*unless*⟩
10.2) ⟨*loop*⟩ → ⟨*anonL*⟩ **while** ⟨*expr*⟩ ⟨*stmts*⟩ ⟨*repeat*⟩
10.3) ⟨*loop*⟩ → ⟨*namedL*⟩ **while** ⟨*expr*⟩ ⟨*stmts*⟩ ⟨*unless*⟩
10.4) ⟨*loop*⟩ → ⟨*namedL*⟩ **while** ⟨*expr*⟩ ⟨*stmts*⟩ ⟨*repeat*⟩

along with the new productions

$\overline{13}$) ⟨*anonL*⟩ → ϵ
$\overline{14}$) ⟨*namedL*⟩ → **id**

Productions 11 and 12 are replaced by the productions

$\overline{11}$) ⟨*unless*⟩ → **unless** ⟨*expr*⟩
$\overline{12}$) ⟨*repeat*⟩ → **repeat**

Productions 10.1–10.4 can then be substituted in production 4.

The cost of altering G is the expansion of the grammar. The additional chain reductions can be eliminated during grammar modification if further production-duplications are acceptable. Clearly, the transformations can be performed automatically. However, the size of the parse tables will increase. If chain-rule elimination is not applied, the changes in G do not alter the behavior of the parser except for the extra chain reductions. Chain-rule elimination speeds up the parser by reducing the number of parsing actions required.

3.4.2 Using Yield-States

Rather than folding rule-disambiguation into the parser, the second approach summarizes derivation sequences using state values. An additional piece of information, called the **yield-state**, is calculated as an attribute of each nondenotative nonterminal in the concrete grammar. The yield-state distinguishes among the various derivation sequences represented by that nonterminal. Yield-states are only needed when the right-hand side of a production in P_{non} contains nonterminals in P_{non} that have alternatives, or when the left-hand side has alternatives.

The yield-state value is kept on an auxiliary stack that is parallel to the parse stack. When a production in the concrete grammar is reduced, a new state value for that production is calculated. The value of the new state depends only on the rule being reduced and the values of the states associated with nonterminal symbols on the right-hand side.

For productions in P_{non}, the yield-state is computed from the yield-states of the nonterminal symbols on the right-hand side of the production. For productions in P_{sig}, the particular production in the abstract grammar to use will be defined by the yield-states of the nonterminal symbols on the right-hand side of the production in the concrete grammar.

$(1, 1')$	(1)
$(2, 2')$	(2 ⟨*stmts*⟩ ⟨*stmt*⟩)
$(3, 3')$	(3 (7 if ⟨*expr*⟩ then ⟨*stmts*⟩) (9))
$(3, 4')$	(3 (7 if ⟨*expr*⟩ then ⟨*stmts*⟩) (8 else ⟨*stmts*⟩))
$(4, 5')$	(4 (10 (13) while ⟨*expr*⟩ ⟨*stmts*⟩ (11 repeat)))
$(4, 6')$	(4 (10 (13) while ⟨*expr*⟩ ⟨*stmts*⟩ (12 unless ⟨*expr*⟩)))
$(4, 7')$	(4 (10 (14 id) while ⟨*expr*⟩ ⟨*stmts*⟩ (11 repeat)))
$(4, 8')$	(4 (10 (14 id) while ⟨*expr*⟩ ⟨*stmts*⟩ (12 unless ⟨*expr*⟩)))
$(5, 9')$	(5 (13) begin ⟨*stmts*⟩ end)
$(5, 10')$	(5 (14 id) begin ⟨*stmts*⟩ end)
$(6, 11')$	(6 exit (13))
$(6, 12')$	(6 exit (14 id))

Figure 3.8: Correspondences between Concrete and Abstract Syntax

The functions for computing yield-states and for selecting productions in the abstract grammar can be computed at language-definition time. The results can be embedded in tables that are consulted during reduce actions in the parser.

Many yield-states will play no role in choosing alternatives. In most cases, the function to compute a new yield-state (or production in the abstract grammar) will either return a single fixed value, or else will consult only one or two positions on the stack. In those cases, careful table layout and packing will speed up tree-building.

3.4.3 Computing Yield-States

This section presents an algorithm for assigning yield-state calculation functions to productions. Derivations are represented using nested rule applications. Only productions in P_{non} appear in the derivations. Their left-hand side is replaced by their rule number in the grammar. For instance, using the grammars of Figure 3.7, the derivation

$$\begin{aligned} \langle stmt \rangle &\underset{3}{\Rightarrow} \langle \textit{if-stmt} \rangle \\ &\underset{7}{\Rightarrow} \text{if } \langle expr \rangle \text{ then } \langle stmts \rangle \langle \textit{else-part} \rangle \\ &\underset{8}{\Rightarrow} \text{if } \langle expr \rangle \text{ then } \langle stmts \rangle \text{ else } \langle stmts \rangle \end{aligned}$$

is denoted by

(3 (7 if ⟨*expr*⟩ then ⟨*stmts*⟩) (8 else ⟨*stmts*⟩))

Figure 3.8 shows the derivations of interest resulting from the grammars given in Figure 3.7.

There are two related tasks here:

1. To calculate, for each rule in P_{non}, a function from the yield-states on its right-hand side to the yield-state for that rule, and

2. To calculate, for each rule in P_{sig}, a function from the yield-states of its right-hand side to the tree-building actions associated with that rule.

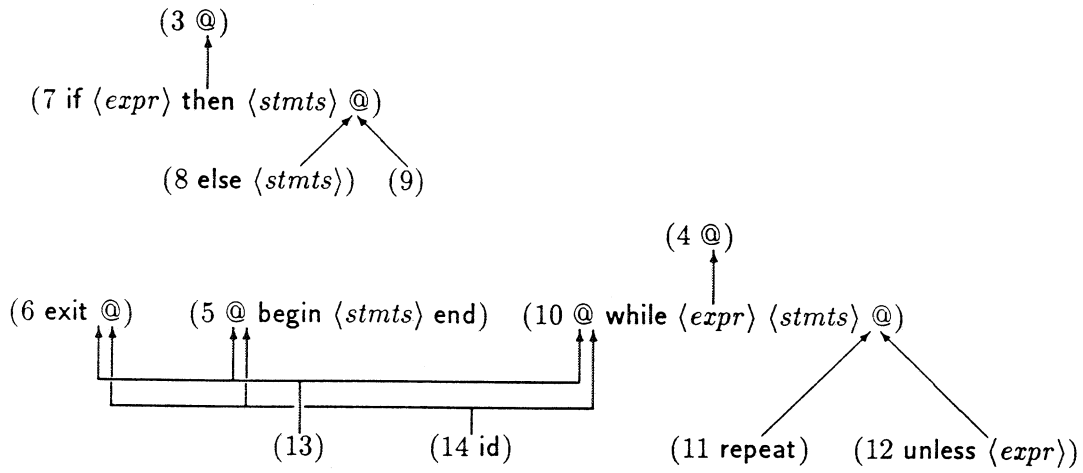


Figure 3.9: Dependency Graph for Derivations of Figure 3.8

To calculate the state-transition functions, first build a dependency graph encoding all possible uses of subderivations. Each node in the graph represents a rule in the concrete grammar, while each edge denotes a possible use of that rule. Figure 3.9 shows the interesting subgraphs of the dependency graph for the running example.

Each node in the graph corresponds to a production in the concrete grammar, and is represented using a pattern having variables denoted by the symbol "@". A pattern variable indicates a nondenotative nonterminal symbol in the production associated with that node in the graph.

Let $DG = (Nodes, Edges)$ be a directed acyclic graph where the edge $\langle n_i, n_j.k \rangle \in Edges$ indicates that position k of node n_j depends on n_i. Nodes are labeled by their corresponding productions in the concrete grammar, for instance n_i is the node associated with production i in the concrete grammar. For instance, in Figure 3.9, the upper subgraph involving the if-then-else construction is specified by

$$DG = (\{ n_3, n_7, n_8, n_9 \}, \{ \langle n_7, n_3.1 \rangle, \langle n_8, n_7.1 \rangle, \langle n_9, n_7.1 \rangle \})$$

Let *States* be an unbounded set of state values, and let *DerivSeq* be a set of derivation sequences. Each edge $e \in Edges$ will be assigned a set of values $val(e) \subset States \times DerivSeq$. The algorithm presented below calculates the assignment.

For each node n, let $pat(n)$ be the pattern associated with n. If $pat(n)$ has m variables, let $pat(n)\langle d_1, \ldots, d_m \rangle$ be the derivation resulting from the substitution of d_i for the i^{th} variable in $pat(n)$.

Define

$$in\text{-}edges(N) = \{ \langle n_j, N.k \rangle \in Edges \text{ for some } n_j, k \}$$

and

$$out\text{-}edges(N) = \{ \langle N, n_j.k \rangle \in Edges \text{ for some } n_j, k \}.$$

Let

$$V_i(N) = \bigcup_{\langle n_j, N.i\rangle \in Edges} val(\langle n_j, N.i\rangle)$$

be the set of values assigned to all edges incident to the i^{th} variable of node n.

Denote the projection on the i^{th} coordinate of $\langle c_1, \ldots, c_m\rangle$ by $\downarrow i$, that is,

$$\langle c_1, \ldots, c_m\rangle \downarrow i \equiv c_i \quad \text{for } 1 \leq i \leq m.$$

The projection operation is extended to sets in the usual way.

The algorithm in Figure 3.10 is a standard DAG scheduling algorithm, with two minor exceptions. First, the values transmitted along the edges are sets of pairs $\langle s, d\rangle$ where s is a yield-state and d is the derivation corresponding to s. Second, the algorithm "looks ahead" along edges to ensure that for any position in a node, all states assigned to any edge terminating at that position are distinct. Looking ahead is handled by the calculation of the set *Out*. The calculation allows values from the set *States* to be used multiple times within a graph, provided that each use is unambiguous. Reusing state values allows an implementation to restrict yield-state values to a relatively small set of values.

The yield-state functions f_n can be simplified by omitting $V_i(n)$ from the calculation whenever $|V_i(n)| \leq 1$.

3.5 Incremental Parsing

The definition of weak $\langle H, \theta_0\rangle$-abstraction provides the foundation for adapting the incremental parsing algorithm of Jalili and Gallier [62] to use an internal representation that corresponds closely to the abstract syntax rather than the concrete syntax.

A full parse tree provides three items of information at each subtree: the state of the parser, the symbol representing the root of that subtree, and the children of the node (if any) as represented in the concrete syntax. This is not the case with an arbitrary internal representation. However, when the abstract syntax is described by a grammar that is a weak $\langle H, \theta_0\rangle$-abstraction of the concrete grammar, it is possible to associate with each node of the internal representation (i) the state of the parser when that node was created, (ii) the nonterminal symbol that represents the node in the concrete grammar, and (iii) an expansion template for mapping the node back to its representation in the concrete syntax.

Incremental LR parsing requires that two subproblems be solved. First, the incremental parser must be able to recreate its state at a point just prior to the first parsing action that might change because of an editing action. Second, the parser must be able to determine which portions of the parse tree have not changed despite the editing action.

A full parse tree contains the record of all parsing actions within its own structure. If a parse tree is to support incremental parsing, each node in the tree must also contain information that describes the state of the parser when that node was created. One way to recreate the state information is to place, in each interior tree node, the parser state that was on top of the state stack (after popping the symbols on the right-hand side of the production) when the node was created. This is the state that was consulted by the **goto** function to compute the new state following the reduction. The initial state of the parser

```
— Calculate the yield-state transition function f_n for each node n.
(∀e ∈ Edges)(val(e) := ∅).
ToDo := Nodes
while (ToDo ≠ ∅)
      let n ∈ ToDo be any node such that
         ∀e ∈ in-edges(n), val(e) ≠ ∅
      if (out-edges(n) = ∅)
         —n is a tree-building node
         then
         foreach ⟨⟨s_1, d_1⟩, ..., ⟨s_m, d_m⟩⟩ ∈ V_1(n) × ... × V_m(n)
                  if (pat(n)⟨d_1, ..., d_m⟩ is a derivation
                                          corresponding to production P
                                          in the abstract grammar)
                    then
                         f_n(s_1, ..., s_m) := build-tree(P)
                    else
                         error
                    endif
         endloop
      else
         Out := ⋃_{(n,n_j.k)∈ out-edges(n)} V_k(n_j) ↓ 1
         Val := ∅
         if (in-edges(n) = ∅)
                  then
                  Let s' ∈ States − Out be a new state
                  f_n() := s'
                  Val := { ⟨s', pat(n)⟩ }
         else
                  foreach ⟨⟨s_1, d_1⟩, ..., ⟨s_m, d_m⟩⟩ ∈ V_1(n) × ... × V_m(n)
                    Let s' ∈ States − Out be a new state
                    f_n(s_1, ..., s_m) := s'
                    Val := Val ∪ { ⟨s', pat(n)⟨d_1, ..., d_m⟩⟩ }
                    Out := Out ∪ { s' }
                  endloop
         endif
         foreach (e ∈ out-edges(n))
                  val(e) := Val
         endloop
      endif
      ToDo := ToDo − { n }
endloop
```

Figure 3.10: Algorithm for Calculating State Transition Functions

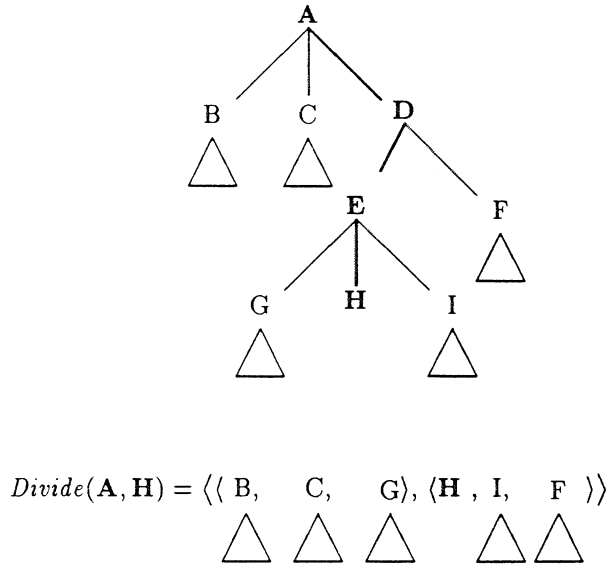


Figure 3.11: *Divide*(**A**,**H**)

is placed in the node representing the root of the entire tree. This state information is used to reconstruct the parser's stack and to help determine whether a subtree can be reused without reparsing.

For an LR(1) parser, the new parse is identical to the original parse up to the point at which the last symbol in the unchanged prefix was shifted. (With a single symbol of look-ahead[1], this is the first point where any symbol in the modified string can affect a parsing action.)

Suppose that the parser's stack is to be reconstructed to the point just prior to when the terminal symbol n_{leaf} was shifted. The *Divide* operation reconstructs the parser's state stack. *Divide* splits the tree along a path from the root to n_{leaf}. The parser states associated with nodes to the left of the path (terminals and nonterminals alike) are pushed onto the growing parse stack. Nodes to the right of the path, representing potentially unchanged subtrees, are retained for later input to the algorithm. Figure 3.11 illustrates the *Divide* operation.

More formally, suppose that the parse stack is to be reconstructed to the point just before the leaf node n_{leaf} was shifted. Suppose that n_i is a node having m children $n_{i_1}, \ldots, n_{i_j}, \ldots n_{i_m}$ and that n_{i_j} is the next node in the path to be divided. If s_{i_j} is the state of the parser stored in node n_{i_j}, then $s_{i_1} \ldots s_{i_{j-1}}$ were the top $j-1$ items on the parse stack when n_{i_j} was parsed and $n_{i_{j+1}}, \ldots, n_{i_m}$ represent the roots of subtrees whose parse must be reconsidered. By walking from the root to n_{leaf} and pushing, for each node n_i, the states $s_{i_1} \ldots s_{i_{j-1}}$ onto the parse stack, the contents of the parse stack can be reconstructed. If the

[1]LR(1) or LALR(1) parsing is assumed throughout this discussion. The extension to k symbols of lookahead is straightforward.

ParseStack: A stack of subtrees (represented by their root nodes). The states and symbols of the nodes in *ParseStack* form the actual contents of the parser's stack.

Input: A stack of subtrees ready to be read.

yield(t) A function that returns the frontier of a tree t.

push(), *pop*(): The usual stack functions extended to sequences.

first(), *rest*(): The usual functions on sequences.

OT: The original parse tree.

w': The modified input string. Assume $w' = a_1 ... a_m$ and that a_{j+1} is the first modified symbol in w'.

Remainder(w') : A function that returns the yet-to-be-parsed suffix of w'.

ParseList, *ReadList*: Variables for node lists returned by subtree operations.

Figure 3.12: Definitions for Incremental Parsing Algorithm

right-siblings $n_{i_{j+1}}, \ldots, n_{i_m}$ are pushed (right-to-left) onto the input stack, then the future inputs to the parser will be retained in the correct order. Subtrees to the left of the path are pushed onto the tree-building stack. The final node in the path n_{leaf} is added to the input stack.

Altogether, four operations on subtrees are defined:

1. *Divide*(t, a_i) takes a subtree t and terminal symbol a_i in the yield of t, and returns the left- and right-hand components of the tree on the path to a_i.

2. *UndoReductions*(t) is the special case of *Divide*(t, a_i) when a_i is the rightmost symbol in the yield of t. Only the left-hand (parse stack) component is returned. This case corresponds to undoing all of the reductions performed when the symbol following a_i was the lookahead token.

3. *Delete*(t, a_i) is used when the initial substring $a_0 \ldots a_i$ was deleted from yield of t. Only a right-hand (input) component is returned. The computation of this component is the same as with *Divide*.

4. *Replace*(t) is the special case of obtaining an input component by dividing the subtrees on the leftmost path from t to a_0. Every subtree on this path has the same parser state in its state information.

Figures 3.12 and 3.13 show the algorithm. In the Jalili-Gallier algorithm, symbols that were deleted during editing must be retained in the parse tree until re-parsing. Newly inserted symbols must also be available to the parser.

3.5.1 Adapting the Jalili-Gallier Parsing Algorithm

The Jalili-Gallier incremental parsing algorithm requires two modifications in order to support incremental parsing using an internal representation closely related to the

```
/* Initialize ParseStack and Input */
(ParseList, ReadList) := Divide(OT, a_k);
push(ParseList, ParseStack);   push(ReadList, Input);
repeat
      case: Remainder(w') begins with an inserted substring
            Parse the new substring.
      endcase
      case: Remainder(w') begins with a deleted substring a_{j_1}...a_{j_k}.
            t := pop(Input)
            if yield(t) was deleted
               then delete t
               else /* some of yield(t) remains */
                    ReadList := Delete(t, a_{j_k});
                    push(ReadList, Input);
               endif
      endcase
      case: Remainder(w') begins with an already parsed substring
            Perform all possible reductions using FirstSymbolOf(Remainder(w'))
            t := pop(Input);
            /* Check the matching conditions */
            if PrecedingStateOf(t) = TopStateOf(ParseStack)
              then if yield(t) was not modified
                        then if the input symbol following yield(t) is changed
                                then /* undo the parse of t */
                                     ParseList := UndoReductions(t);
                                     push(ParseList, ParseStack);
                                else /* t can be re-used */
                                     push(t, ParseStack);
                                endif
                        else /* yield(t) is modified */
            /* Let a_j be the symbol occurring immediately before the first change in yield(t) */
                             (ParseList, ReadList) := Divide(t, a_j);
                             push(ParseList, ParseStack);
                             push(ReadList, Input);
                        endif
              else /* states don't match */
                   ReadList := Replace(t);
                   push(rest(ReadList), Input);
                   a := FirstSymbolOf(first(ReadList));
                   Continue parsing with a as the input until a is shifted
              endif
      endcase
until end of input
```

Figure 3.13: Jalili-Gallier Incremental Parsing Algorithm

abstract syntax. Similar modifications are required to adapt other incremental parsing algorithms to use grammatical abstraction. First, *Divide* must be changed so that at each step on the path from the root to the leaf, the subtrees are expanded to their concrete representations. Second, while reconstructing the parse stack, the parse tables must be consulted to calculate the states that were not stored in the tree.

Suppose $\widehat{G}$ is an $\langle H, \theta_0 \rangle$-abstraction of G where $\widehat{G}$ describes the abstract syntax and G describes the concrete syntax If $\widehat{A} \rightarrow \widehat{\alpha}$ is used in the internal representation created during parsing, then for some $A \in$ *Significant* and some $\alpha \in (\Sigma \cup \mathit{Significant})^*$, where $\theta(A) = \widehat{A}$ and $\theta(\alpha) = \widehat{\alpha}$, there is a corresponding derivation $A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha$ in G. When a subtree whose root node represents the production $\widehat{A} \rightarrow \widehat{\alpha}$ is processed by the incremental parser, either the subtree is divided using the modified *Divide* operation, or the entire subtree can be retained. If the entire subtree is retained, then A should be the nonterminal symbol shifted by the parser.

When a subtree is divided by one of the subtree operations, it is a representation α in the concrete syntax that must actually be processed. This representation α is a single-level expansion template. Expanding $\widehat{\alpha}$ back into α generates symbols in the concrete grammar that may have been erased by the time that subtree was created. The expansion α must be determined from $\widehat{A} \rightarrow \widehat{\alpha}$ and A.

For instance, suppose the subtree representing the conditional statement '$\langle stmt \rangle \rightarrow \langle expr \rangle\ \langle stmts \rangle$' is expanded back into its concrete representation using the grammars $\widehat{G'_1}$ and G_1 of Figure 3.6. The expansion in the concrete syntax is '**if** $\langle expr \rangle$ **then** $\langle stmts \rangle$ **";"**'.

Starting from the parse state that was at the top of the parse stack when the production $A \rightarrow \beta$ was reduced, the symbols in the expansion must be parsed to recover the intermediate state not present in the abstract tree. This processing will not encounter any errors, since it is the reconstruction of a correct parse. No new subtrees will be created at this stage.

For most subtrees $\widehat{A} \rightarrow \widehat{\alpha}$ in the internal representation, the associated nonterminal symbol in the concrete grammar A and the template α can be stored in tables describing the abstraction correspondence rather than in the internal representation tree. These tables can be created at parser generation time; the size of the tables depends on the language and the correspondence, not on the size of the actual internal representation. The exception occurs when the abstraction creates induced chain derivations, or when a single rule in the abstract grammar corresponds to more than one rule in the concrete grammar.

3.5.2 Induced Chain Derivations

Derivations of the form $A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha$ where $A \in \mathit{Significant}, \alpha \in (\Sigma \cup \mathit{Significant})^*$ and $\theta(A) = \theta(\alpha)$ need not be represented structurally. When $|\alpha| = 1$, the absence of a structural representation is not a problem for incremental parsing. Simple chains neither add terminal symbols to the concrete representation, nor do they have any effect on the calculation of parse states. (The parser state to be retained in the subtree is the same for all elements of a simple chain.)

When $\theta(A) = \theta(\alpha)$ and $|\alpha| > 1$, however, there is extra information to be retained; terminal symbols of the concrete grammar may have to be added to the expansion. These

derivations are called **induced chain derivations**. In an induced chain derivation, a single symbol in the abstract grammar appears on the right-hand side of the derivation after the mapping θ is applied.

For instance, consider the LR parse for ⟦(id * id)⟧ using the simple expression grammar of Figure 3.2 when parentheses are abstracted. The abstract syntax is simply '⟨*expr*⟩ → id "*" id' in $\widehat{G'_0}$, but the correct expansion for incremental parsing may have to include the parentheses.

If an induced chain derivation is not represented directly by a subtree of the internal representation, the node that represents the nonterminal $\widehat{A}$ in the abstract grammar must be tagged with an expansion template for generating the terminal symbols of the induced chain derivation in order to generate a correct expansion in the concrete syntax. If the concrete representation provided by the user is to be recreated, there may be a sequence of annotations on a single node in the internal representation; the templates must be applied in the reverse order of their recognition. For instance, a node representing ⟦((((id * id))))⟧ in the internal representation would have four induced chain annotations associated with it if all four levels of parentheses are preserved.

During incremental parsing, if a subtree annotated with induced chains is retained, the nonterminal symbol used to represent the subtree should be the left-hand side of the first induced chain derivation in the sequence of annotations. In the running example, the nonterminal symbol used would be ⟨*factor*⟩, which corresponds to the production ⟨*factor*⟩ → "(" ⟨*expr*⟩ ")". When the subtree is divided, the subtree is expanded to restore the erased symbols. The concrete nonterminal symbol corresponding to the sole child in the internal representation should then be used to represent the child. This nonterminal symbol would be ⟨*term*⟩ in the example (corresponding to the production '⟨*term*⟩ → ⟨*factor*⟩ ⟨*mulop*⟩ ⟨*term*⟩').

A node in the internal representation may have n induced chain annotations associated with it. Using the straightforward approach, that node would have $n + 1$ parse states associated with it—one per induced chain reduction plus one for the original significant production. If the entire sequence of induced chain derivations is expanded during conversion from abstract to concrete then the node requires only a single associated parse state—the state corresponding to the last-recognized induced chain derivation. The state can be captured by destructively updating previous values of the parse state as induced chain derivations are recognized. Expanding the entire sequence of induced chain derivations at once might require extra parsing, but the storage savings achieved by eliminating the additional parse states makes it worthwhile. In practice, the complete expansion will often be required anyway.

3.5.3 Requirements on the Tree Building Component

The tree builder, then, must store or reconstruct information about each node of the internal representation in order to support incremental parsing. This information consists of the state of the parser when the subtree rooted at that node was created, the nonterminal symbol of the concrete grammar to be used to represent the node during incremental parsing, and the expansion template to be used during incremental parsing. It is possible to access both of the latter items through a single index into tables describing

the weak abstraction correspondences.

During parsing, only reductions of productions in P_{sig} are of interest to the subtree creation routines. Suppose $A \rightarrow \beta \in P_{sig}$ is being reduced, and that $A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha$. There are four possibilities:

1. $\theta(A) \neq \theta(\alpha)$ and $|\theta(\alpha)| > 1$. This is the normal case; a new subtree must be created and tagged with the appropriate parse state and index for recovering A and α.

2. $\theta(A) \neq \theta(\alpha)$ and $|\theta(\alpha)| = 1$. This production can be represented either structurally or by an annotation.

3. $\theta(A) = \theta(\alpha)$ and $|\alpha| = 1$. This is a simple chain rule; no tree-building actions are necessary.

4. $\theta(A) = \theta(\alpha)$ and $|\alpha| \neq 1$. This may be an induced chain derivation; if a new subtree is specified by the language designer, it will be created. Otherwise an annotation will be added to the single subtree on the right-hand side. This subtree is available from the tree-building stack. First, the annotation "$A \rightarrow \alpha$" is added to the list of induced chain annotations on the subtree, becoming the "topmost" production. This information retains both the nonterminal symbol in the concrete grammar A and the expansion template α. Second, the parser state associated with the subtree can be destructively updated to be the state that would be retained if a new subtree was being created.

3.6 Strong Grammatical Abstraction

Weak grammatical abstraction allows us to represent the derivation of every sentence in a concrete syntax abstractly. However, it does not allow us to map from an arbitrary sentential form generated from the abstract grammar back to corresponding concrete syntax. In order to do so, there must be at least one concrete version of every sentential form of $L(\widehat{G})$. There may be many such versions; for example, the concrete representation of an abstract expression may contain arbitrarily many levels of parenthesization. In addition, it must be possible to fill in the structural information missing from the abstract syntax such as precedence provided by parentheses.

One can always map back to concrete syntax, provided that (i) each nonterminal in $\widehat{G}$ maps back to a unique significant nonterminal in G and (ii) productions in $\widehat{P}$ correspond to derivations in G. The erased terminals and missing structure represented in the concrete grammar by nondenotative nonterminal symbols can then be reconstructed.

Strong $\langle H, \theta_0 \rangle$-abstraction ensures that the two grammars derive the same language under θ, and provides some of the extra information necessary to implement template-based structure editing in an editor that uses bottom-up parsing.

Definition 6 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar, and let $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ be a reduced context-free grammar. Let *Significant* be as defined previously, and let θ be the extended mapping function. Then $\widehat{G}$ is a **strong $\langle H, \theta_0 \rangle$-abstraction** of G if

1. $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G.
2. The mapping θ restricted to H is invertible, that is, if $\theta(A) = \theta(B)$ and $A, B \in H$, then $A = B$.
3. For every production $\widehat{A} \rightarrow \widehat{\alpha} \in \widehat{P}$, there exists $A \in \textit{Significant}$ such that
   (a) $\theta(A) = \widehat{A}$ and
   (b) There exists a finite derivation $A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha$ where $\alpha \in (\Sigma \cup \textit{Significant})^*$ and $\theta(\alpha) = \widehat{\alpha}$.

   (It follows from this condition that $\theta(H) = \widehat{N}$.)

∎

Again consider the grammars of Figure 3.2. The grammar $\widehat{G_0}$ cannot be a strong $\langle H, \theta_0 \rangle$-abstraction of G_0, since the mapping θ would have to be invertible on H. In this case, the nonterminal symbol $\langle binop \rangle$ in the abstract grammar would have to correspond to both $\langle addop \rangle$ and $\langle mulop \rangle$ of the concrete grammar. However, $\widehat{G'_0}$ is a strong $\langle H, \theta_0 \rangle$-abstraction of G_0.

Theorem 2 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar and let $\widehat{G}$ be a strong $\langle H, \theta_0 \rangle$-abstraction of G. If $\widehat{A} \underset{\widehat{G}}{\overset{*}{\Rightarrow}} \widehat{\alpha}$ where $\widehat{\alpha} \in \widehat{V}^*$ then there exists some $A \in H$ and $\alpha \in (\Sigma \cup \textit{Significant})^*$ such that $A \underset{G}{\overset{*}{\Rightarrow}} \alpha$, $\theta(A) = \widehat{A}$, and $\theta(\alpha) = \widehat{\alpha}$.

To prove Theorem 2, the following lemma is used.

Lemma 1 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar and let $\widehat{G}$ be a strong $\langle H, \theta_0 \rangle$-abstraction of G. If $C, D \in N$ and $\theta(C) = \theta(D)$, then $C \overset{*}{\Rightarrow} x_1 D x_2$ where $\theta(x_1 x 2) = \epsilon$.

Proof of Lemma The proof is by analysis of three cases.

Case 1: $C \in H$ and $D \in H$

The definition of strong abstraction requires that θ be invertible on H, so $C = D$ because $\theta(C) = \theta(D)$.

Case 2: $C \in H$ and $D \in \textit{Identified}$

Since $D \in \textit{Identified}$, $\exists D' \in H$ such that D' is identified with D. By the definition of θ, $\theta(D) = \theta(D')$. Since $\theta(D') = \theta(C)$ and both $C, D' \in H$, the invertibility of θ insures that $D' = C$. By the definition of *Identified*, then, $C \Rightarrow \beta \underset{P_{N-H}}{\overset{*}{\Rightarrow}} x_1 D x_2$ where $\theta(x_1 x_2) = \epsilon$.

Case 3: $C \in \textit{Identified}$

Since $C \in \textit{Identified}$, there is a $D' \in H$ such that $\theta(C) = \theta(D')$ and $C \underset{P_{N-H}}{\overset{*}{\Rightarrow}} y_1 D' y_2$ for some $y_1, y_2 \in \Sigma^*$ where $\theta(y_1 y_2) = \epsilon$. Since $\theta(D) = \theta(D')$, it is also true that either

$D \in H$ and $D = D'$ or else $D \in$ *Identified* and D is Identified with D'. If $D = D'$, take $x_1 = y_1$ and $x_2 = y_2$. Otherwise, ($D \in$ *Identified*) for some β_2, z_1, and z_2, $D' \Rightarrow \beta_2 \underset{P_{N-H}}{\overset{*}{\Rightarrow}} z_1 D z_2$ where $\theta(z_1 z_2) = \epsilon$. In this case, $C \overset{*}{\Rightarrow} y_1 D' y_2$ and $D' \overset{*}{\Rightarrow} z_1 D z_2$ where $\theta(y_1 y_2) = \theta(z_1 z_2) = \epsilon$. Taking $x_1 = y_1 z_1$ and $x_2 = z_2 y_2$ completes the proof.

Proof of Theorem 2 The proof is by induction on the length n of the derivation $\widehat{A} \overset{*}{\Rightarrow} \widehat{\alpha}$ in $\widehat{G}$.
Base: $n = 0$. Follows from the fact that $\theta(H) = \widehat{N}$.
Inductive Argument Assume that the theorem is true for all derivations of length $l \leq n$. Suppose that $\widehat{A} \overset{n+1}{\Longrightarrow} \widehat{\alpha}$.

Then for some $\widehat{B} \rightarrow \widehat{\alpha_2} \in \widehat{P}$ and some $\widehat{\alpha_1}, \widehat{\alpha_3} \in \widehat{V}^*$,

$$\widehat{A} \overset{n}{\Longrightarrow} \widehat{\alpha_1}\widehat{B}\widehat{\alpha_3} \Rightarrow \widehat{\alpha_1}\widehat{\alpha_2}\widehat{\alpha_3} = \widehat{\alpha}$$

By the induction hypothesis, for some $A \in H$ such that $\theta(A) = \widehat{A}$ and $A \underset{G}{\overset{*}{\Rightarrow}} \alpha'$, there exists an $\alpha' \in (\Sigma \cup \mathit{Significant})^*$ such that $\theta(\alpha') = \widehat{\alpha_1}\widehat{B}\widehat{\alpha_3}$. Since θ is a homomorphism, there exist $\alpha_1, \alpha_3 \in (\Sigma \cup \mathit{Significant})^*$ such that $\theta(\alpha_1) = \widehat{\alpha_1}$, $\theta(\alpha_3) = \widehat{\alpha_3}$, and there exists $C \in \mathit{Significant}$ such that $\theta(C) = \widehat{B}$. It remains to show that $\exists \alpha_2 \in (\Sigma \cup \mathit{Significant})^*$ such that $\theta(\alpha_2) = \widehat{\alpha_2}$ and $C \underset{G}{\overset{*}{\Rightarrow}} \alpha_2$.

By case 3 of the definition of strong abstraction, corresponding to the production $\widehat{B} \rightarrow \widehat{\alpha_2}$ there is a nonterminal symbol $D \in \mathit{Significant}$ and a finite derivation $D \Rightarrow \delta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha_2$ where $\alpha_2 \in (\Sigma \cup \mathit{Significant})^*$ such that $\theta(D) = \widehat{B}$ and $\theta(\alpha_2) = \widehat{\alpha_2}$. Although $\theta(C) = \theta(D) = \widehat{B}$, there is no guarantee that $D = C$. But Lemma 1 assures us that $C \overset{*}{\Rightarrow} x_1 D x_2$ where $\theta(x_1 x_2) = \epsilon$. So

$$A \overset{*}{\Rightarrow} \alpha_1 C \alpha_3 \overset{*}{\Rightarrow} \alpha_1 x_1 D x_2 \alpha_3 \overset{*}{\Rightarrow} \alpha_1 x_1 \alpha_2 x_2 \alpha_3$$

where $\theta(\alpha_1 x_1 \alpha_2 x_2 \alpha_3) = \theta(\alpha_1 \alpha_2 \alpha_3) = \widehat{\alpha}$.

Corollary 2 If $\widehat{G}$ is a strong $\langle H, \theta_0 \rangle$-abstraction of G, then $L(\widehat{G}) \subseteq \theta(L(G))$. **Proof:** The definition of weak $\langle H, \theta_0 \rangle$-abstraction and case 2 of the definition of strong $\langle H, \theta_0 \rangle$-abstraction assure that if $A \in \mathit{Significant}$ and $\theta(A) = \widehat{S}$, then $S \overset{*}{\Rightarrow} x_1 A x_2$ where $\theta(x_1 x_2) = \epsilon$. The remainder of the proof follows from Theorem 2.

Corollary 3 If $\widehat{G}$ is a strong $\langle H, \theta_0 \rangle$-abstraction of G, then $\theta(L(G)) = L(\widehat{G})$.
Proof: Follows from Corollaries 1 and 2.

3.7 Top-Down Tree Construction

Syntax-directed editors, such as MENTOR [38] or the Cornell Program Synthesizer [130], construct tree-structured documents from the top down. Starting from the root,

the document is elaborated by successively replacing unexpanded placeholders in the tree with templates that represent new substructures. For instance, a ⟨*stmt*⟩ placeholder might be expanded into a subtree that represents a **while** loop. The resulting subtree may contain new placeholders. In the **while** example, new placeholders for ⟨*expr*⟩ and ⟨*stmts*⟩ will appear as illustrated in Figure 3.14.

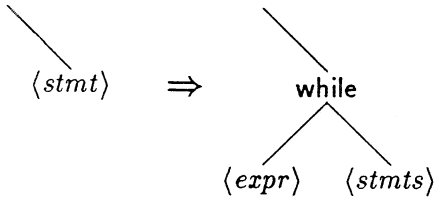


Figure 3.14: Replacement of ⟨*stmt*⟩ with a While-loop Construct

Syntactic correctness of the tree is maintained by allowing only legal subtree replacements.

Weak grammatical abstraction allows a language-manipulation system to support textual manipulation by providing incremental parsing while using the abstract syntax to define the internal representation of a linguistic object. Yet there are times when a top-down, structure-oriented approach is desirable.

The problem of top-down tree construction in a syntax-recognizing editor is that of creating trees that are syntactically valid and that can support incremental parsing. This problem is harder than in the syntax-directed approach because the trees that are built top-down must be identical to the tree constructed from the same concrete representation using bottom-up parsing. Thus when a placeholder is expanded (in the internal representation), the concrete grammar must be consulted. In particular, the correspondences between productions of the abstract grammar and derivations in the concrete grammar established by strong $\langle H, \theta_0 \rangle$-abstraction will be used. The trees constructed top-down will be identical to those created by a bottom-up parsing of the same concrete form.

Theorem 2 states that if $\widehat{G}$ is a strong $\langle H, \theta_0 \rangle$-abstraction of G, then every derivation in $\widehat{G}$ has a corresponding derivation in G. This is the result needed to insure that each internal representation can support incremental parsing. In order to parse incrementally, each node of the internal representation must be decorated with the same information saved by the tree-building component of the bottom-up parsers: parse states, concrete nonterminal symbols, expansion templates, and the results of induced chain derivation reductions.

The nonterminal symbols and the expansion templates are the symbols and forms of the concrete grammar corresponding to the productions in the abstract grammar used to derive the tree. Placeholders in the tree (representing unexpanded subtrees) are nonterminal symbols in the abstract grammar. To expand a placeholder $\widehat{A}$ using the production $\widehat{A} \rightarrow \widehat{\alpha}$, some analysis is needed. The analysis may result in the addition of induced chain annotations to the subtree representing the desired expansion.

In *Pan*, this analysis is handled by creating tables at language-definition time that tell, for any pair of nonterminal symbols, whether one symbol can be replaced with another.

If the replacement is legal, the tables also encode whether induced chain annotations must be added. These tables can also be used in a batch mode to annotate internal representations created by other programs.

An unexpanded placeholder must have a corresponding representation in the concrete grammar to be processed during incremental parsing. The nonterminal symbol used is specified by the inverse of the mapping function θ. Suppose that a placeholder $\widehat{A}$ is being expanded, and that $\theta(A) = \widehat{A}$ for some $A \in H$. If the incremental parser can shift the placeholder because its yield and its following symbol are unchanged, then the placeholder can use A as its concrete representation. However, if the context of a placeholder has been textually altered, this approach is insufficient.

Consider an unexpanded placeholder $\langle expr \rangle$ in an internal representation. This nonterminal symbol in the abstract grammar may have to be represented by the nonterminal symbol $\langle factor \rangle$ in the concrete grammar to ensure correct parsing. Using the definitions of θ and strong abstraction, the information that allows the system to choose the symbol $\langle factor \rangle$ to represent the placeholder can be computed. In some cases, however, the top-down constructor must insert induced chain annotations in order to insure that the modification will be correctly parsed.

3.8 Implementation

Strong grammatical abstraction provides the theoretical base for the implementation of parsing and tree building used in *Pan I*. The formal notions of abstraction are used to establish requisite correspondences between the abstract grammar and the concrete grammar. Annotations on the productions of the abstract grammar provide further information about the internal tree. When several productions of the abstract grammar are represented using a single operator in the abstract syntax—hence a single node in the internal representation—the annotation mechanism is used to preserve the information required for incremental parsing.

In a LADLE description, the language designer provides two views of the abstract syntax: as a context-free grammar and as a collection of operators defining the internal representation. The abstract grammar can be regarded as the set of representation rules for operators in the abstract syntax. Each operator in the abstract syntax is associated with one or more productions in the abstract grammar. Because the abstract grammar is viewed as a collection of representation rules, the productions in that grammar provide a place to embed pretty-printing directives.

LADLE performs the checking required to ensure that the abstract grammar is truly an abstraction of the concrete grammar. Extended BNF constructs are handled by converting them to simple context-free grammar productions using standard grammatical transformations. The results presented above, therefore, extend naturally to EBNF-described languages.

Some productions in the abstract grammar should not be explicitly represented as operators. The writer of a language-specification can specify which productions in the abstract grammar will not be represented by operators in the internal representation. Those productions are limited to be simple chain productions, in which the right-hand side is a

single symbol. This case arises often when the abstract syntax is described by a context-free grammar; it was discussed in connection with the first example (Figure 3.2, page 40).

Pan I uses a modified incremental parsing algorithm. After implementing the Jalili-Gallier algorithm described above, it was noted that a modification to the parsing tables, coupled with a change to the way subtrees were decomposed for reparsing, would allow us to parse incrementally without having to retain the parser state in each node of the internal representation. The parse tables are modified by calculating parser actions when nonterminal symbols in the concrete grammar are lookahead symbols. Not retaining the parse state further shrinks the internal representation and simplifies top-down elaboration at the cost of enlarged parse tables.

To date, language descriptions have been written for Modula-2, ASPLE, "C", COLANDER and for LADLE's own language description language. In the Modula-2 description, the concrete grammar contains 161 productions, and 82 nonterminal symbols, while the abstract grammar contains 157 productions and 79 nonterminals. The total number of symbols appearing on the right-hand side of productions in the concrete grammar is 371, while in the abstract grammar it is 217. Only 97 of the productions in the abstract grammar are represented structurally in the internal tree; another 44 productions may appear as annotations.

Chapter 4

Contextual Constraints and Logical Constraint Grammars

Contextual constraints play an important role in programming languages. A **contextual constraint** expresses a condition imposed by the static semantics of a language. A program or document is considered well-formed when all of its contextual constraints are satisfied. Given a description of a target language, *Pan* can check the contextual constraints imposed by that language.

Target languages are described to *Pan* using a *logical constraint grammar* (LCG). A logical constraint grammar associates goals expressing constraints of the target language with structures in the abstract syntax of the target language. A document is considered well-formed whenever all of its associated goals are satisfied.

Logical constraint grammars apply logic programming and consistency maintenance to the problem of specifying and checking contextual constraints. Read as a declarative specification, a logical constraint grammar defines the contextual constraints of the language as a collection of propositions. Read procedurally, a logical constraint grammar specifies a program for the enforcement of the constraints. Logical constraint grammars provide a complete descriptive mechanism for specifying contextual constraints. Goals and rules in a logical constraint grammar are written using clausal logic.

The implementation of a LCG-based system includes a logic-based programming language, a database system, a consistency manager, and a means to express contextual constraints and to gather information about a document. An entire logical constraint grammar description can be analyzed statically to extract information required for efficient incremental evaluation. The presence of a static analyzer simplifies descriptions and relieves the authors of language descriptions from specifying many details.

This chapter introduces logical constraint grammars and discusses their semantics. Section 4.1 presents the general notion of textual analysis problems and develops a set of desirable characteristics for systems that solve textual analysis problems as part of an integrated environment. Following that, Section 4.2 surveys research related to the specification of static-semantic constraints. Section 4.3 reviews PROLOG-based logic programming. A detailed definition for logical constraint grammars appears in Section 4.4. Finally, Section 4.5 discusses logical constraint grammars and their relationship to other methods for specifying

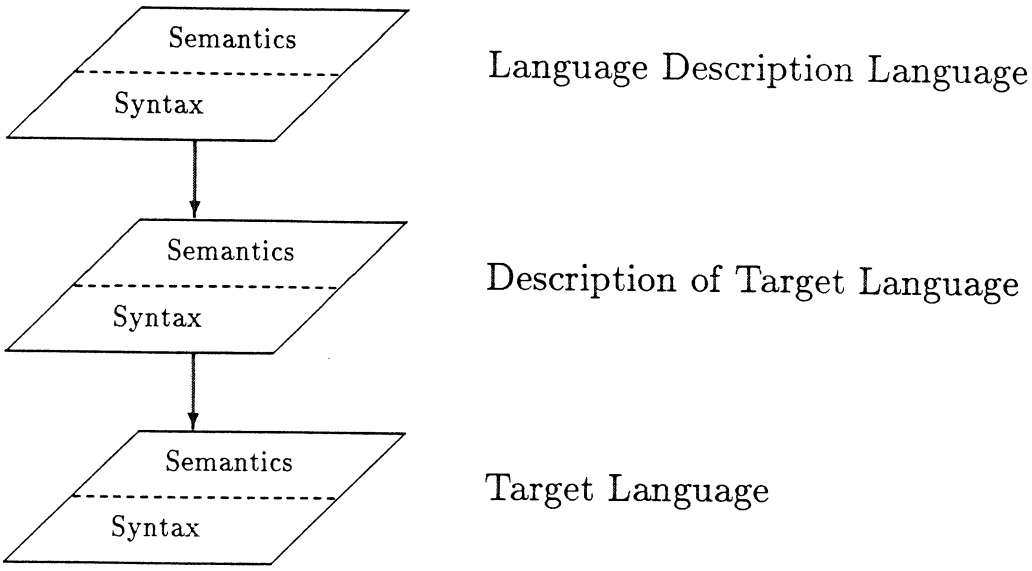


Figure 4.1: The Description Language, the Description, and the Language Described

static semantics.

A COLANDER description of contextual constraints forms one component of a LADLE description for a target language. COLANDER is presented in Chapter 5. COLANDER descriptions can be analyzed statically in order to gather required by the consistency manager. Chapter 6 discusses the static analysis performed on a COLANDER description. It also describes the process of evaluating a logical constraint grammar in order to enforce the constraints that the logical constraint grammar specifies.

Figure 4.1 reviews the relationships that were established in Figure 1.1 (page 7) among a target language, a language description, and a language description language. The *target language* is the language being described, while the *language description language* is the language used to write the description. Recall that the interpreter for the *language description* executes the *dynamic* semantics of the language description language to enforce the *static* semantics of the target language.

4.1 Textual Analysis Problems

Verifying that contextual constraints are satisfied is one of a larger class of problems called textual analysis problems. A **textual analysis problem** is problem that can be solved by statically analyzing the source (text) of a program or document. Parsing, program slicing [138], data flow analysis, the analysis performed on a LCG description on behalf of the consistency manager (Chapter 6), and the kinds of analysis found in the Programmer's Apprentice [116, 136], Masterscope [86], or Microscope [7, 79] are all textual analysis problems.

Solving a textual analysis problem can be viewed as the creation, maintenance, and

manipulation of a collection of data relating different portions of a document's contents. For instance, checking whether an identifier in a program has been declared at the point of use relates two portions of the source text: the use and the declaration. Determining whether the assignment of a value to a variable in a program is legal involves not only fragments of the program text but also axioms defining what it means for a variable to be assignable in that language. Maintaining the information needed to display a call graph or to answer queries like "What functions call this function?" also fits nicely into this model.

Many textual analysis problems can be expressed in terms of constraints (or actions) associated with structures in the abstract syntax of a language. Such problems include checking the contextual constraints imposed by a language as well as the kinds of analysis performed by tools such as Masterscope [86] or Microscope [7, 79]. Logical constraint grammars provide support for efficiently solving these kinds of textual analysis problems.

Methods used in interactive software development environments for solving textual analysis problems should have the following characteristics:

- **Clear descriptive model.** The underlying description technique should be simple to explain and use. Simplicity is necessary for both the writer and the reader of descriptions. The correctness of the description should be readily apparent.
- **Clean semantic model.** The semantic model for calculating the solution should be simple and effective.
- **Incremental evaluation.** The method must easily handle incremental changes in the document being analyzed. Minor changes to the document should trigger the least recomputation necessary to update the solution. Supporting incremental changes to the language description is less necessary, albeit desirable in some cases.
- **Incremental update of maintained data.** Along with incremental evaluation, the data maintained by the evaluator should be modifiable incrementally. By way of contrast, in some attribute-grammar based systems, the entire symbol table is used as the value of a single attribute. If this value is recreated from scratch after a modification, too much is being done.
- **Useful to other tools.** The data gathered and analyzed should be readily available to all tools that could use it, including the editor/browser itself. For instance, "semantics-directed" editing commands should be easy to write.

How well logical constraint grammars meet these characteristics is discussed in Section 4.5.

4.2 Related Work

This section discusses other approaches to the description and enforcement of contextual constraints in language-based editors. Some of the approaches, particularly attribute grammars, have been applied to a wide variety of textual analysis problems.

4.2.1 Attribute Grammars

Attribute grammars [75, 76] are a popular method for specifying and evaluating contextual constraints[1]. This section reviews the use of attribute grammars in language-based editors, their strengths and their weaknesses. It begins with an informal definition of an attribute grammar. Then, research concerning attribute grammars that is related to this dissertation is discussed.

Informal Definition

A formal definition of an attribute grammar is unnecessary for the purpose of this dissertation. Instead, those aspects of attribute grammars and attribute-grammar based formalisms that are used when comparing logical constraint grammars to attribute grammars will be presented informally. Informally, an **attribute grammar** is a context-free grammar in which

- Each nonterminal symbol is augmented by a set of attributes. Attributes are either *inherited* or *synthesized*. Inherited attributes are used to pass information down a derivation tree generated by the grammar towards the leaves; synthesized attributes are used to propagate information up the tree towards the root.
- Each production in the context-free grammar is augmented with semantic functions and predicates that depend only on the values of attributes associated with symbols appearing in that production. Semantic functions compute values for the attributes that receive values at that production. The result of a semantic function can depend only on the values of its arguments. Because semantic functions are purely functional, the result of a function cannot change unless the value of one of its arguments changes.

An *attributed syntax tree* is a syntax tree (defined by the underlying context-free grammar) that is decorated with attribute values as defined by the semantic functions.

Inherited attributes associated with the symbol on the left-hand side of a production together with synthesized attributes associated with symbols appearing on the right-hand side of that production designate values obtained from the surrounding tree. They serve as input values to semantic functions. Inherited attributes associated with symbols on the right-hand side of the production are passed down to the children; synthesized attributes associated with the symbol on the left-hand side of the production are passed up to the parent.

The *evaluator* for an attribute grammar assigns values to the attributes associated with each subtree in the syntax tree by executing the semantic functions defined for that subtree. A semantic function can be executed whenever all of its input arguments have received values. In general, the evaluator visits each subtree in the tree one or more times, executing the semantic functions and propagating the attribute values computed at that subtree to the parent or children as necessary.

The strength of the attribute grammar formalism is in its declarative semantics and in the ease with which incremental evaluators can be constructed. Changes in attribute

[1]The recent work by Deransart, Jourdan and Lorho [33] reviews the theory and practice of attribute grammars. It includes a catalog of existing systems.

values signal clearly the need to re-evaluate the semantic functions that used the changed values; propagation of changes through the tree follows apace. Much of the early work in attribute grammars focussed on defining efficient evaluation strategies for various classes of attribute grammars.

However, attribute grammars appear to be unwieldy in practice, being difficult to read, write, and evaluate. The attribute grammar formalism itself usually specifies only the attributes and their interrelationships. A different formalism is used to specify the semantic functions and the data types used in the description. Most often, that formalism is a close cousin to an imperative programming language such as Pascal or C. Much of the interesting information in a attribute-grammar based specification for a system resides in the code that implements the semantic functions and data types. Attribute grammars are claimed to be a declarative formalism, but the declarative portion of a description extends only to attribute flow and dependencies, not to the subtleties of the language or system being defined. In contrast, the logical constraint grammars provide a single complete description mechanism for specifying contextual constraints that is open to static analysis.

Attribute grammars are deficient in two other ways. First, attribute values always flow along paths defined by the syntax tree itself. If a value is to be used at some point far from where it is computed, each subtree on the path from the point-of-definition to the point-of-use must propagate that value by using a "copy rule." While evaluators can implement copy-rules efficiently, it is the responsibility of the writer of the attribute grammar to see that attributes values are propagated correctly. The values propagated around the tree can often be large—entire symbol tables or environments. The size causes problems for efficiently updating such values. Hoover's work [57] addressed this issue.

Second, the entirety of the attribute values constitute a closed database; there is no easy way to make information in the attribute values available to other tools. Horwitz and Teitelbaum [59] addressed this problem by placing attribute values into a relational database.

Incremental Evaluation

Reps [32, 111, 115] and Ganzinger *et. al* [45], among others, provide approaches to incremental evaluation of attribute grammars.

Reps' results include algorithms for incremental attribute evaluation [115]. His general approach is to maintain *superior* and *inferior* characteristic graphs for each subtree in the attributed tree. These graphs represent dependencies among attributes appearing at that subtree. After a modification to an attributed syntax tree, incremental evaluation proceeds by "undoing" previous analyses and then updating the attributes affected by the modification. The characteristic graphs determine which attributes are affected. Evaluation terminates once all attributes affected by a change have been re-evaluated. These algorithms are optimal in the number of attributes re-evaluated for a given change. Their drawbacks are in the management of the dependency graphs and in the general problems of attribute grammars.

Ganzinger *et. al* [45] outline an incremental evaluation algorithm in which a summary of update information is maintained. This summary information—"preserved," "updated," "not-known"—guides the update process. When attributes are scheduled for re-

evaluation, local decisions are made by using a "view" that summarizes global dependency information. The views could be the inferior/superior graphs, or could be other models of dependence. The algorithm itself accepts a view as a parameter.

Non-local Attribute Dependencies

Attributes have two basic uses: as local information (for example, type checking in expressions) and as global information (for example, maintaining consistency of declarations and uses). Underlying attribute evaluation is the assumption that all attributes are propagated along edges of the syntax tree. The problem is that many subtrees may lie between the definition of an attribute value and its use. The requirement for attribute values to flow along paths defined by the syntax tree is a major problem with attribute grammars. Explicitly propagating the attribute through the tree complicates the description and can slow the evaluator. *Non-local attribute dependencies* avoid this inefficiency by directly linking remote attribute occurrences.

To handle non-local attribute flow, Johnson and Fischer [63] require explicit declarations for the sets of interacting remote attributes. These sets are called "context sensitive relation sets" (CSRS). CSRS are used to propagate attribute values over links between attributes that are separate from the syntax tree. Algorithms for incremental evaluation using CSRS are used in the POE system at the University of Wisconsin. One drawback to this approach is that the author of the attribute grammar must specify the CSRS explicitly. This potentially error-prone process complicates the task of writing a specification.

Reps and his coworkers [112] attempt to handle the problem of non-local dependencies automatically within the attribute grammar evaluator. By restricting the dependencies that can exist between the attributes of two nonterminal symbols, they are able to define a *covering relation* that allows the evaluator to order the evaluation of the attributes associated with those symbols. Their technique handles multiple subtree replacements (generalizing the simple structure-editing approach) and aggregate values as well as non-local dependencies.

Beshers and Campbell [17] address the problem of non-local attribute dependencies by adding user-definable data structures that are maintained by the attribute evaluator. The designer of the data structure is required to define explicitly the modifications required when various actions, including retraction, occur. Modifications to the data structures are monitored by their evaluator in order to trigger re-evaluation. Their solution is used in the SAGA editor.

Kaiser [68] also uses a data-structure based approach to non-local data dependencies. A non-local dependency is handled by a combination of a data structure that allows the author of a language description to gather together similar structures (e.g., all of the uses of a variable) and to propagate events directly to the semantic functions that use that data structure.

Sharing Attribute Values with Other Tools

Horwitz and Teitelbaum [59] address the need for general access to the information gathered by an attribute grammar by combining attribute grammars with relational

databases. Attribute values are stored in a relational database. The attribute grammar and its evaluator provide incremental evaluation. The relational database provides data management, query access, and multiple views.

Two new kinds of database relations are needed to make this possible. *Locally-derived relations* depend upon the (editing) state of the document. *Unconstrained relations* are independent of the editing state. Database views are used to group related data. For instance, the collection of declarations defined in a single scope of a program can be treated as a view of the underlying database.

Horwitz and Teitelbaum had to extend the relational database model to accommodate their application. Operations involving transitive closure, order-dependent processing, and arithmetic operations were added. They also addressed the problem of incremental view update. Linton [84] encountered many of the same problems when he embedded the entire text of a program into a relational database.

Other Extensions

Kaiser [68] extends the attribute grammar model to handle dynamic semantics. *Action equations* are equations that are embedded in an event-driven architecture. They play the role of attribute functions in a attribute grammar. Events trigger the evaluation of equations in the same way that user operations can trigger attribute re-evaluation. When an event occurs, the evaluator orders the equations triggered by that event prior to re-evaluation. The algorithm for determining the ordering is based on Reps' algorithm for incremental attribute evaluation.

Ganzinger and Hanus [46] combine attribute grammars with logic programming in order to modularize the components of a compiler. Their approach generalizes attribute grammars (slightly) in order to implement the semantic specification as a PROLOG program. This, in turn, is a subcase of Deransart and Maluszynski's *logical attribute grammars* [34]. A logical attribute grammar can be transformed into a PROLOG program. Attributes in the original description become logical variables in the PROLOG program. Neither Ganzinger and Hanus nor Deransart and Maluszynski are concerned directly with incremental evaluation.

ERGO [82] is a system for rapidly prototyping environments to support formal program development. In response to the requirements of ERGO, the ERGO Attribute System [100] was developed. It extends the attribute grammar paradigm to handle inter-program structure sharing, demand-driven attribute evaluation, improved incremental update, and to better handle histories of attribute values.

4.2.2 Action Routines

Action routines [90, 91] are arbitrary procedures that are associated with the productions of a context-free grammar. When an editing event occurs, the code associated with that event is called. Action routines are therefore extremely flexible and powerful. Their drawbacks include the arbitrariness of the paradigm and the need for the author of the routines to deal with all aspects of incrementality. For instance, routines that modify data structures are usually required to supply an inverse routine. ALOE [91], DOSE [69], and PECAN [108] use action routines as their basis for contextual constraint checking. Kaiser [68]

extends action routines to handle dynamic semantics using a method similar to action equations.

4.2.3 PSG

PSG [13] supports contextual analysis of arbitrary incomplete program fragments, including the detection of errors within a fragment. This is accomplished by the use of *context relations*. A program fragment is represented by an attributed syntax tree. Each program fragment is summarized by its context relation. A fragment is considered valid if it can be embedded into a correct (larger) fragment. A context relation summarizes the set of all "still-possible attribute values"—the values that have not been ruled out by the evaluation of other context constraints. An empty context relation indicates that there is no possible valid assignment of values to the attributes of the fragment, and therefore indicates an error.

Context relations are composed using the *natural join* operation of relational database theory. The natural join operation is implemented using unification over a many-sorted algebra. The paper by Bahlke and Snelting [13] describes their incremental analysis algorithm.

While context relations work well for simple languages such as Pascal or mathematical logic, the approach does not handle languages with more complicated name spaces like Modula-2 or Ada, or languages that support operator overloading or type polymorphism.

4.2.4 Natural Semantics and TYPOL

Natural semantics [66] is a logic-based formalism based on Plotkin's structured operational semantics [103]. In natural semantics, a language description specifies an abstract syntax together with axioms and inference rules that characterize the structures in the abstract syntax. The collection of axioms and inference rules is identified with a logic akin to *natural deduction* [105]. Reasoning about the target language is reduced to proving theorems in that logic. Tree pattern matching is used to determine which rules apply in any given case.

TYPOL [37, 66], the semantic description language used in CENTAUR [20], is based on natural semantics. A TYPOL description is a collection of axioms and inference rules. The general evaluation strategy is to compile the rules of the description into PROLOG rules and the resulting single equation to be proved into a PROLOG goal.

Natural semantics and TYPOL alone provide an elegant descriptive mechanism for many kinds of contextual constraints. However, the original PROLOG-based implementation was inefficient since it was not incremental. Attali [10, 11] addressed the problem of incremental evaluation in TYPOL by transforming a TYPOL definition into an attribute grammar. Both the TYPOL definition and the resulting attribute grammar use the same abstract syntax. (The transformation from a logic program to an attribute grammar presented by Deransart and Maluszynski [34] uses the proof trees underlying the logic program to define the syntax tree used by the attribute evaluator.) Attali [11] also shows how to implement partial evaluation of TYPOL programs and how to extend the approach to dynamic semantics.

While the transformation from TYPOL to an attribute grammar achieves incremental evaluation, the attribute values manipulated by the attribute grammar are not necessarily modifiable incrementally. This means that without care, the entire symbol table for a program will be treated as a unit unless techniques like Hoover's are adopted. Moreover, by adopting attribute grammars, one adopts not only their strengths but their weaknesses.

4.2.5 Visibility Control Rules

Attribute grammars and action routines are primarily oriented toward efficient implementations of the static-semantics of a language. Research by Reiss [107], Wolf [141], and Garrison [47] focused on descriptive techniques for defining the scope and visibility rules found in current languages. In general, the hope has been that a clear descriptive technique could be automatically translated into an efficient implementation. Garrison describes the work by Reiss and by Wolf. This section reviews Garrison's work, since it is related to our own.

In Garrison's terminology, a *name* is a symbol that can be appear in the external (concrete) syntax of the language. *Entities* represent linguistic objects—objects in the underlying semantics. A *binding* associates a name with an entity. *Visibility control rules* define

- which names are legal in a language description,
- how names are associated with entities,
- how the creation of a new binding affects the visibility of other bindings, and
- what should happen when the rules are violated.

The *search model* and the *visible set model* are two different approaches to describing visibility control rules. The search model describes how to locate a binding when given a name. The visible set model (and the related *range model*) describes the set of all bindings in effect at any given point in a document.

Both the search model and the visible set model can be implemented using an *inheritance graph*. An inheritance graph represents the dependencies between bindings within a program. A vertex in the inheritance graph represents a *visibility region*—a region in the program or document having the same set of visible bindings throughout. An edge in the inheritance graph represents the inheritance of visibility from one region to another. Edges are directed. At each vertex, the inherited visible bindings can be filtered or combined with other inherited or locally-generated bindings. Error detection occurs as bindings are processed at a vertex. The resulting set of visible bindings may then be propagated to other visibility regions. A complete specification of the visibility control rules for a language defines how each construct in the language affects the construction of the inheritance graph for a document.

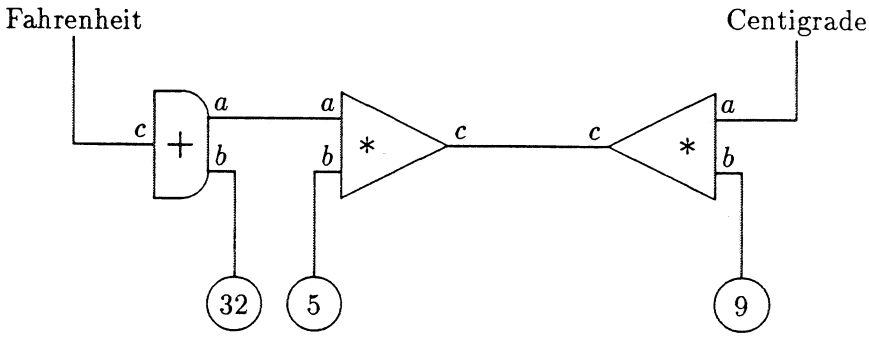


Figure 4.2: Constraint Equation Network for Converting Temperatures

4.2.6 Constraint Equations

Other researchers [19, 64, 83, 131] have considered constraint-oriented programming in the domain of discrete combinatorial problems. This work deals with satisfaction of networks of *constraint equations* as occur in circuit emulation, scheduling, layout, or linear programming. Constraint equations are not directly relevant to the research reported here. However, similarities in terminology and technique, in particular the use of logic programming and consistency maintenance, creates confusion between LCGs and constraint equations.

Figure 4.2 appeared in Steele's dissertation [64]. It shows a simple constraint equation for converting temperatures between Fahrenheit and Celsius. The equation is represented as a network. This network has two leads labeled "Fahrenheit" and "Centigrade." Asserting a value on the "Fahrenheit" lead causes the corresponding value to appear on the "Centigrade" lead, and vice versa. Each arithmetic node in the network is bidirectional: for instance, the node labeled "+" representing addition can be interpreted as either $c = a + b$ or $a = c - b$ or $b = c - a$, while the nodes labeled "$*$" representing multiplication can be interpreted as either $c = a \times b$ or $a = c/b$ or $b = c/a$. Dependency-directed backtracking is used to update the solution to the constraint network when the given values are changed.

Van Hentenryck [131] investigates extensions to the search mechanisms of logic programming in order to better support constraint satisfaction. He models constraint satisfaction as a search problem; improved search strategies improve the performance of constraint satisfaction programs.

Other researchers, including DeRemer and Jullig [35], Kaplan [71, 72], Rosselet [119], and Vorthmann [133] have proposed ways to specify and enforce contextual constraints. Their primary concerns differ from our own, so their contributions are not summarized here.

4.3 Logic Programming

This section reviews first-order logic, logic programming, and PROLOG as needed for this study.

Constants:	a, b, and c (possibly subscripted)
Variables:	u, v, w, x, y, and z (possibly subscripted)
Functions:	f, g, and h (possibly subscripted)
Predicates:	P, Q, and R (possibly subscripted)

Figure 4.3: Notation Pertaining to Logic Programming

4.3.1 Definitions

The definitions here are those of first-order logic as used in logic programming. They are adapted from those given by Lloyd [85]. Figure 4.3 describes the notational conventions for logic that are used throughout the remainder of this dissertation.

Definition 7 A **term** is defined inductively by:

(a) A constant is a term.

(b) A variable is a term.

(c) If f is a n-ary function and $\mathsf{t}_1, \ldots, \mathsf{t}_n$ are terms, then $\mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ is a term.

When a term does not contain any instances of a variable it is called a **ground term.** Terms that are not ground terms are called **nonground.** ∎

Definition 8 A **(well-formed) formula** is defined inductively as:

(a) If P is a n-ary predicate and $\mathsf{t}_1, \ldots, \mathsf{t}_n$ are terms, then $\mathsf{P}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ is a formula (called an **atomic formula** or simply an **atom**).

(b) If F and G are formulas, then so are $(\neg F), (F \vee G), (F \wedge G), (F \rightarrow G)$, and $(F \leftrightarrow G)$.

(c) If F is a formula and x a variable, then $(\forall x F)$ and $(\exists x F)$ are formulas.

One may write $(G \leftarrow F)$ for $(F \rightarrow G)$. ∎

Definition 9 A **literal** is an atomic formula or the negation of an atomic formula. ∎

Definition 10 In many discussions of logic programming, terms of the form $\mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ and atomic formulae are classified together. A **functor** is either an atomic formula of the form $\mathsf{P}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ or a term of the form $\mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$. The predicate or function name is the **name** of the functor and the terms $\mathsf{t}_1, \ldots, \mathsf{t}_n$ are called the **arguments** of the functor. ∎

Definition 11 An **expression** is a term, a literal, a conjunction of literals, or a disjunction of literals. ∎

Definition 12 A **clause** is a formula of the form The predicate or function name is the **name** of the functor and the $\forall x_1 \ldots \forall x_k (L_1 \vee \ldots \vee L_n)$ where each L_i is a literal and $x_1, \ldots x_k$ are all of the variables occurring in $(L_1 \vee \ldots \vee L_n)$. For clauses of the form $\forall x_1 \ldots \forall x_k (A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n)$ where the A_i and B_j are atomic formulae and $x_1, \ldots x_k$ are all of the variables occurring in those formulae, the notation "$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$" will be used. Note that the commas in the antecedent $B_1, \ldots, B_n$ denote conjunction while the commas in the consequent $A_1, \ldots, A_m$ denote disjunction. The quantifiers are implicit. ∎

Definition 13 There are four categories of clauses.

1. A **program clause** is a clause of the form "$A \leftarrow B_1, \ldots, B_n$" ($n \geq 1$). The positive literal A is called the **head** of the clause, and $B_1, \ldots, B_n$ is called the **body** of the clause.
2. A **unit clause** is a clause of the form "$A \leftarrow$ ", that is, a clause consisting of a single consequent having no antecedents.
3. A **goal clause** is a clause "$\leftarrow B_1, \ldots, B_n$" ($n \geq 1$) having an empty consequent.
4. A clause having neither an antecedent nor a consequent is called the **empty clause** and is denoted by "□".

∎

Definition 14 A logic program P is a collection of program, goal, and unit clauses. ∎

Logic programming is modeled on refutation theorem proving. To prove that

$$\exists y_1 \ldots \exists y_r (B_1 \wedge \ldots \wedge B_n) \tag{4.1}$$

is a consequence of a program P, the negation of the formula is added to the axioms specified by P and then one attempts to derive a contradiction. If a contradiction can be derived, then the original formula must be a logical consequence of P.

When the formula in Equation 4.1 is negated, the goal clause "$\leftarrow B_1, \ldots, B_n$" is obtained. Working top-down, a logic program derives successive goals. A contradiction has been obtained if the system is able to derive the empty clause.

Unification [117] is the heart of an interpreter for a logic program. A unification algorithm attempts to match two expressions t_1 and t_2 by finding a *substitution* θ for the variables appearing in t_1 and t_2 such that $t_1\theta = t_2\theta$.

Definition 15 A **substitution** is a finite set of pairs of the form $X_i = t_i$ where X_i is a variable and t_i is an expression (that is, a term, a literal, or a clause) such that $\forall j \neq i, X_j \neq X_i$ and X_i does not appear in any t_j for any j. The result of applying a substitution θ to an expression T, denoted $T\theta$ is the expression obtained by replacing every instance of the X_i in θ with its corresponding t_i. ∎

A unification algorithm computes the most general substitution that will unify two expressions t_1 and t_2.

If the current goal is "$\leftarrow C_1, \ldots, C_k$", a single step in the derivation consists of selecting one of the C_j, unifying it with the head of some program clause "$A \leftarrow B_1, \ldots, B_n$" or unit clause "$A \leftarrow$" to obtain a substitution θ, and generating a new goal

$$\leftarrow (C_1, \ldots, C_{j-1}, B_1, \ldots, B_n, C_{j+1}, \ldots, C_k)\theta$$

This step is repeatedly applied until the empty clause is derived or until no further substitutions are possible. Unifying a literal with a unit clause results in a goal containing fewer literals.

Substitutions are generated using unification. In a logic programming system, if the proof succeeds, then the final substitution is formed by composing the intermediate substitutions. The final substitution defines a collection of bindings between the variables appearing in the original goal and the expressions to which those variables were ultimately bound. Logical variables therefore have the property that they start out unbound, and can be gradually defined. Gradual definition occurs when a variable is bound to a value that contains other unbound variables. Once a variable is bound to a ground expression, it remains bound to that expression.

4.3.2 PROLOG

PROLOG has become the standard language for logic programming. In PROLOG, programs are expressed using clausal logic. A clause has the general form "`A :- B, C, D.`" and may be read declaratively as

> "The literal `A` is true if the literals `B` and `C` and `D` are all true."

or procedurally as

> "To prove `A`, first attempt to prove `B`; if `B` is provable attempt to prove `C`; and if `C` is provable attempt to prove `D`."

The literal `A` is called the *head* of the clause, and the literals `B, C, D` are called the *body* of the clause. In the terminology of Definition 13, a PROLOG *fact* is a unit clause, a *goal* is a logic goal, and a *rule* is a program clause.

In PROLOG, variables are denoted by names beginning with upper-case letters, brackets denote lists, and vertical bars denote list construction. For example, the PROLOG clauses

```
append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).
```

define a relationship `append`(X, Y, Z) between three lists X, Y, and Z that holds whenever Z is the result of appending Y to X.

PROLOG places all clauses into an ordered database. The database is searched in the order that items were added to it. Clauses can be added at either the beginning (**asserta**) or the end (**assertz**) of the database.

4.4 Logical Constraint Grammars

Logical constraint grammars (LCGs) adapt logic programming and consistency maintenance to the problem of specifying, checking, and maintaining contextual constraints. Logic programming provides a natural approach to this problem. First, the context-sensitive aspects of a language are often phrased using some informal approach to logic. Formalizing the informal definition using clausal logic is therefore relatively straightforward. Second, the act of checking contextual constraints can be viewed as satisfying the constraints relative to

some collection of information. In pass-oriented applications like compilers, this collection of information is represented by a symbol table.

When the collection of information persists as the document changes, the collection must be updated incrementally. A consistency manager maintains consistency between the information in the collection and the current state of constraint enforcement.

The development of logical constraint grammars draws together many of the threads mentioned above: an orientation toward logic as seen in ERGO, PSG, and natural semantics; the use of an explicit database; the separation of concerns between visibility rules and other aspects of contextual constraints. Logical constraint grammars are not as proof-oriented as natural semantics. Instead, by providing consistency maintenance on a data repository, they support efficient incremental evaluation while freeing the writer of a description from the details required by other models.

Definition 16 A **logical constraint grammar** is a triple $\langle G, \mathcal{L}, \mathcal{DB} \rangle$ where

1. $G = (V, \Sigma, P, S)$ is a context-free grammar.
2. $\mathcal{L}$ is a collection of rules and goals expressed in a logic programming language. The goals in $\mathcal{L}$ can be associated with a production or terminal symbol in G, or can be independent of any production or terminal symbol in G.
3. $\mathcal{DB}$ is a database in which each element is a $\langle label, tuple \rangle$ pair. A *label* identifies a collection of tuple values. A *tuple* is a unit clause (data value) that may contain *labels*. The set of all $\langle label, tuple \rangle$ pairs identified by the same *label* is called a **collection of tuples**, or just a **collection.**
4. Each *instance* of a production or terminal symbol in G is associated with a specific collection of tuples in $\mathcal{DB}$. This collection is called the **context** for the subtree representing that production or nonterminal symbol.

∎

The context of a subtree is determined dynamically as the LCG is evaluated. In an LCG, unlike a PROLOG program, there may be many goals to be satisfied. As will be discussed, the ordering among goals is not formally specified, so standard PROLOG programming techniques that rely upon known orderings among tuples (data values) in the database may not apply. In particular, the use of **assert** and **retract** in a LCG differs from their use in PROLOG.

Figure 4.4 shows a very simple LCG for a language that requires that each name be defined before it is used. (This example is much too simple for any real programming language.) The description assumes that $\mathcal{DB}$ contains a single collection of tuples. Assume that the terminal symbol "id" has an associated character string that is the textual representation of that identifier. The notation used resembles COLANDER. The prefix "*?*" is used to denote logical variables.

The example contains two rules, defining what it means for a name to be visible—a name is visible whenever it is bound or has been imported. Each rule matches a single fact, either "bound(*?name*)" or "imported(*?name*)". The goal associated with the use of a name accesses the actual string name associated with the identifier to ensure that the name

Grammar:

1 ⟨*program*⟩ →⟨*def*⟩* ⟨*use*⟩*
2 ⟨*use*⟩ → "USE" id
3 ⟨*def*⟩ → "DEF" id

Two rules:

visible(*?name*) :- bound(*?name*).
visible(*?name*) :- imported(*?name*).

Goal associated with production 2:

:- string-name($id, *?name*), visible(*?name*).

Goal associated with production 3:

:- string-name($id, *?name*),
not(visible(*?name*)),
assert(bound(*?name*)).

Figure 4.4: Simple Logical Constraint Grammar

is visible. The goal associated with a definition checks to see whether the name is already defined, and, if not, adds a tuple "bound(*?name*)" to the database.

An evaluator for an LCG description applies the description to a document represented by a syntax tree. Each subtree in the syntax tree is described by a production in the context-free grammar underlying the LCG. The **goals associated with a subtree** in the syntax tree are the goals associated with the production in the context free grammar that describes the subtree.

Definition 17 An **evaluator for a logical constraint grammar** associates the goals of an LCG description with subtrees in the syntax tree that represents a document, and then attempts to satisfy the goals associated with each subtree in the syntax tree. A document is considered well-formed whenever all of the goals associated with that document have succeeded. ∎

Evaluating a LCG consists of visiting subtrees and attempting the goals associated with those subtrees. In general a goal can execute rules, can create new collections, and can add new tuples to the database. Every collection and tuple in the database was created during the successful evaluation of some goal. When a goal associated with a subtree in the syntax tree succeeds, any collections or tuples created by that goal are said to be *owned* by that subtree.

An incremental evaluator for a LCG ensures that all of the goals that can be satisfied are satisfied, and that any unsatisfiable goal remains unsatisfied. The evaluator for a LCG first attempts to execute those goals that are independent of any syntactic structure. These goals are used to initialize the database. After attempting the structure-independent goals, the structure-dependent goals are attempted. The evaluator completes processing whenever all of the goals are successfully proved, or no further goals can be proved.

Incrementally evaluating a LCG requires an interpreter for the logic language and a consistency manager that monitors changes to the database and reattempts goals as

necessary.

Data values (tuples) are be added to the database as a side-effect of satisfying a goal. Values remain in the database until the goals that created them are retried and generate new values, or until the subtree that owns them is removed from the syntax tree. This means that the database fits the "closed world assumption" [110]—if a goal fails, then its negation is assumed to be true.

During the evaluation of a LCG, satisfying some goals requires satisfying other goals. For instance, one goal may assert a data value that will be required by another. Many interactions between goals act through the database. In other cases, locally-determinable flow of contexts or other data from one subtree to another is available for determining order dependencies. The evaluator of a LCG may use, but does not require, knowledge of dependencies between goals. Naturally, if the user of an LCG system understands the evaluation strategies employed by the system, or if the evaluator makes use of dependence analysis, the performance of a given language description can be improved.

Since the order that goals are evaluated is unspecified, the author of a LCG cannot assume any known ordering among data values within a collection. This contrasts with normal PROLOG-style logic programming, in which the programmer can make use of the known ordering of clauses in the PROLOG database.

4.4.1 Goals and Rules

A logical constraint grammar description of a target language is used to impose contextual constraints on syntactic structures in a document. Each structure in the document corresponds to a production in the context-free grammar underlying the LCG. The constraints on a structure are expressed by goals associated with the corresponding production.

Every goal associated with a syntactic structure is defined relative to a collection of data called a **context datapool**. The context datapool for each syntactic structure is determined by the author of the language description; usually, it is the same as the context datapool of the parent structure. Goals can create and propagate context datapools in addition to expressing constraints of the target language. Context datapools are ordinarily used to pass information down the syntax tree, from parent to child. For example, in a typical programming language, the context datapool may contain the information necessary to find the bindings defined in the current scope.

Clearly separating the context from the goals in a logical constraint grammar helps the author of a language description to focus on the essentials of each. Goals are defined relative to context datapools; context datapools must contain the information necessary to satisfy or disprove their goals. The primary use for a context datapool is to provide access from the syntax tree to a more general database of information; the secondary use is to propagate information locally from structures to substructures in the syntax tree. Propagation of context datapools is similar to methods developed for the ERGO system [100].

Rules express the basic definitions of the target language, in addition to the basic definitions used by the language description. A typical structure-independent rule helps to define type similarity in a strongly-typed language such as Modula-2.

The literals in a LCG description are categorized as either procedures or data. The

category of a literal is determined by its name. Literals that are matched against procedures are called **procedure literals**; literals that are matched against values in the database are called **data literals.**

For instance, in the definition of append given by

append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).

append would be considered a procedure name, whereas in PROLOG the first clause would be called a "fact". The distinction between procedures and data is useful for the analysis described in Chapter 6.

4.4.2 Collections

The logic programming language used to express the goals and rules of a LCG must provide ways to create and manipulate collections of tuples. In contrast, a PROLOG database is a monolithic, ordered collection of clauses. The database required for textual analysis is generally a directed graph (often a tree), where collections are the nodes in the graph and references to collections are the edges. Graphs could be flattened into a PROLOG-style database by adding extra arguments to the values stored in the database. Such an approach is used in the (common) example of lineage, where a PROLOG programmer represents family trees using the notion of "parent" as a functor name, and maintains the tree structure explicitly.

It is useful to distinguish three kinds of collections, each holding its own kinds of data. **Datapools** are collections of **facts**. Typically, datapools contain large numbers of facts, and are used to represent scopes when describing programming languages. **Entities** are collections of **entity properties**. Entities tend to be small; typically they are used to hold the attributes of a particular program object such as a variable or a procedure. **Subtrees** can also be considered as collections holding **subtree properties**. In COLANDER, subtrees are created and destroyed during syntactic analysis.

Datapools

A *datapool* is a container for *facts*. Datapools are used to collocate facts that can or should be treated as a single unit. A context datapool is just a special kind of datapool. For example, each scope in a program might be represented using a separate datapool that contains facts about the declarations appearing in that scope, along with data relating that scope to the other scopes in the program. Datapools are nodes in a graph-structured database. Facts within a datapool may refer to other datapools.

Figure 4.5 illustrates a typical use of datapools. The syntax tree for a simple program is shown in the center of the figure. The program consists of procedure ⟦ P ⟧ that declares two variables ⟦ X ⟧ and ⟦ Y ⟧, and a second procedure ⟦ Q ⟧ that also declares a variable ⟦ X ⟧.

Arrows formed by dotted lines indicate the connection between subtrees in the syntax tree and context datapools. The subtree representing the procedure ⟦ P ⟧ uses context

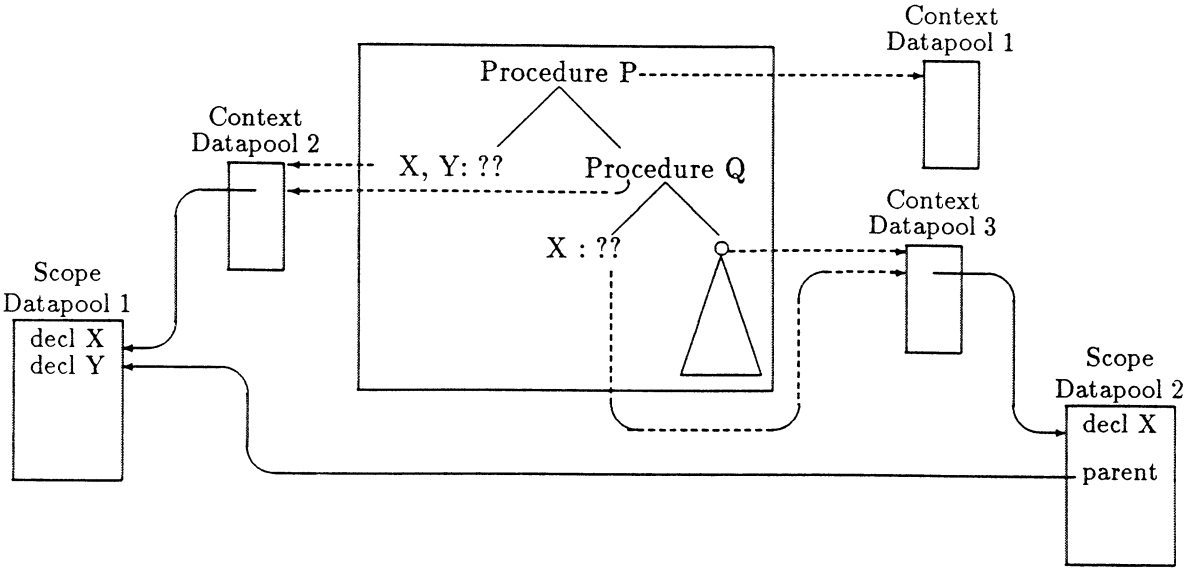


Figure 4.5: Using Datapools to Represent Scopes

datapool 1. Its children, the declarations of ⟦ X ⟧, ⟦ Y ⟧, and ⟦ Q ⟧ receive and share context datapool 2. The children of the subtree representing ⟦ Q ⟧ share context datapool 3.

Context datapools 2 and 3 contain facts that refer to datapools representing the scopes for ⟦ P ⟧ and ⟦ Q ⟧ respectively. Facts representing declarations are placed in those datapools rather than the context datapool. (Separating the context in the derivation tree from the current scopes is a useful technique when writing complex language descriptions.) Additionally, scope datapool 2 contains a fact "parent" that refers back to scope datapool 1. This reference can be used to access the enclosing scope.

By partitioning the database into datapools, individual datapools can be used to represent the various collections of data in a document being edited. Datapools could be represented in a flat database by adding an extra argument to every value stored in the database. However, there are two drawbacks to such an approach.

First, flattening the datapools into the database may be inefficient unless the database manager knows precisely which functor arguments distinguish the various collections. Satisfaction of a conjunctive request is quadratic in the size of the database; satisfying $A \wedge B$ may require one to look at n^2 facts where n is the number of facts in the database. By maintaining and searching only the appropriate path through the relevant datapools the search time can be lessened.

Second, if distinct datapools represent distinct objects in the underlying document, then it should be easy to manipulate them. Thus there should be mechanisms for establishing, manipulating, and changing the organization of the database. Sandewall [121] has argued that the organization of the database should itself be one of the manipulable objects held by the database. In a logical constraint grammar description, the organization

of the database is described by the data in the database.

Subtree Annotation

It must be possible to denote and annotate subtrees in a LCG description. Subtree annotations are called **subtree properties**. A subtree property is a named value associated with a subtree.

Subtree properties are used to transmit information locally through the tree. For instance, within an expression tree, each subtree might be annotated with the type of the expression that the subtree represents. Computing the type of a subtree requires access to annotations on its children. Local structural information can be represented using system-maintained subtree properties. The ability to designate local subtrees (such as parent, children, left-sibling) and their annotations is essential to expressing contextual constraints.

Representation of Objects in the Target Language

An **entity**[2] is a collection that can be used to represent objects of the target language. Information about entities is cast as named values called **entity properties**. It must be possible in an LCG description to allocate and annotate entities.

4.4.3 The Logic Database

The logic database manipulated by goals in a LCG description is composed of (i) datapools containing facts, (ii) subtrees having properties, and (iii) entities having properties. The *values* in the database are either facts or property values.

Values can be dynamically asserted and retracted by the evaluator and consistency maintenance mechanisms. Dynamically changing procedures—procedures that change either by altering their definitions or by letting scope rules affect their definitions—cause major problems for consistency maintenance. Dependency information would have to be retained for procedures as well as data, and the analysis of language descriptions currently performed at language-definition time would have to be extended to handle run-time alterations of the description. There are several reasonable assumptions to make about procedures, the simplest being that they are known at language-definition time and they don't change.

Subtrees in the syntax tree **own** the collections and values created by their associated goals during evaluation.

Values remain in the database until the goals that asserted them are retried and generate new values, or until the subtree that owns them is removed from the syntax tree. Once created, collections are permanently associated with their owning subtree. The values associated with a collection may change repeatedly, but the collection itself retains its existence and identity until its owning subtree is destroyed. Collection are destroyed only when their owning subtree is removed from the syntax tree.

[2]This differs from Garrison's use of the term, but is in accord with his basic model. An entity in a LCG description generally represents one of Garrison's entities in a target language.

4.5 Discussion

Logical constraint grammars are directed toward solving textual analysis problems concerned with constraint satisfaction and data acquisition. Other textual analysis problems, such as parsing or dataflow analysis, are best performed using specific algorithms. Although it is possible to cast solutions for other problems onto the LCG mold, overgeneralizing this approach causes unnecessary complexity. Good partitioning of concerns in the front-end to an integrated development environment leads to different methods for describing the static- and dynamic- aspects of languages. Moreover, well-tailored algorithms and data structures outperform universal approaches.

It is possible to emulate an attribute grammar directly using a LCG However, such an approach ignores the strengths of LCGs without necessarily improving the overall description. Attribute grammars require attribute values to follow paths determined by the syntax tree. Logical constraint grammars discover data dependencies automatically as they propagate through a database that includes, but is distinct from, the syntax tree.

It is the existence of this database that allows LCGs to extend beyond checking contextual constraints. The interpreter for the logic language allows full play of its power in performing deductions about the document being edited. Other tools can use client interfaces to the database to access information. The logic programming language itself provides an inference engine that can be used to extract and manipulate the data stored in the logic database.

Using contextual mechanisms like datapools to partition the database of a program is not new[3]. Datapools appeared in AI languages like PLANNER [128], Conniver [88], and QA4 [120], and more recently have been used in Worlds [42, 93] and assumption-based truth maintenance systems [30]. In all of these cases, datapools capture and manipulate the dynamic state of a problem solver. As the problem solver investigates new portions of a search space, datapools are used to represent new aspects of the state space incrementally. If the new state is later discarded, it is easy to restore the prior state of the search from the contents of the datapool. Information is usually propagated from parent to child, leading to a hierarchical structure among datapools. In contrast, the datapools used in COLANDER are more persistent, since they can be used to represent the (more static) name spaces of documents as well as contextual information. Moreover, the connections between datapools of an LCG description need not be strictly hierarchical.

Meta-Prolog [12, 21] uses a datapool-like construct in order to create new logical theories incrementally. Each theory is a complete logic program; conversely a logic program represents a specific logical theory. Multiple theories allow a user to define alternative approaches or evaluation strategies within a single system. The structure of the system of theories becomes an object of interest in its own right.

An LCG is a meta-level description of a program or document, but the underlying implementation techniques are not themselves "higher-order". In fact, the logic-programming language implemented by COLANDER does not include many of PROLOG's primitives for dissecting and constructing PROLOG code. The combination of incremental parsing with LCG evaluation obviates any requirement for second-order unification to as-

[3]The paper by McDermott [89] is a good review of the use of datapools in traditional AI applications.

sociate rules with syntactic structures. Instead, the goals of an LCG are associated directly with structures in the abstract syntax that are detected during parsing. The association between the syntax and the goals is quite close, making the development of a new language description more difficult than is desirable in an environment focussed on the development of new languages. However, since the target applications are for support for document development rather than language development, this rigidity is acceptable. In return, the implementation of the system can be more efficient.

How do logical constraint grammars address the desirable characteristics for solutions to textual analysis problems given on page 67? The logic-programming basis for a LCG provides a clean, simple semantic model possessing both a declarative and and procedural interpretation. Incremental evaluation is addressed by the presence of multiple goals, each expressing either a contextual constraint or an action required to establish state within a language description. Changes in the logic database can force a goal to be retried. Incremental update of data is provided by the explicitly structured database together with consistency maintenance. Client interfaces which provide access to the the database can easily be established so that other tools can access and manipulate the data in the database. Thus, logical constraint grammars provide all of the characteristics previously outlined.

COLANDER is the implementation of a LCG evaluator for use with *Pan* and LADLE. In addition to goals, procedures, data values, entities, datapools, and context datapools, COLANDER provides memoized-procedures, database triggers, and ways to communicate error messages to a user of *Pan*. The next chapter presents the COLANDER language. Consistency maintenance and incremental evaluation of COLANDER descriptions is presented in Chapter 6. Chapter 7 discusses a LCG description for the programming language Modula-2.

Chapter 5

Colander

COLANDER is the component of *Pan* that implements logical constraint grammars. *Pan* uses COLANDER to enforce contextual constraints and provide "semantics-oriented" editing operations and viewing. The implementation of COLANDER led to several extensions to the PROLOG model for logic programming. The use of a structured rather than monolithic database together with the need for consistency maintenance forced many of the changes. The PROLOG paradigm has not been restricted except in minor ways. This chapter introduces the concepts and notation used to write a language description to be processed by COLANDER. A basic knowledge of PROLOG [25, 126] is assumed.

The COLANDER language is a sublanguage of the LADLE language description language. It is PROLOG-like in its syntax and semantics, but extends PROLOG in several ways. The COLANDER language provides the facilities required to describe a logic constraint grammar: rules, goals, a structured database composed of multiple datapools, subtree denotation and annotation, and entity creation, manipulation, and annotation. In addition, the COLANDER language provides memoized procedures for computing and remembering the values of subtree properties, facilities for communicating with the editor user, and database triggers for activating goals when a datum is added or removed from the database.

This chapter describes the facilities provided by COLANDER. Appendix A summarizes the language.

"*Tiny*" is a simple language similar to many actual programming languages. The syntax for *Tiny* is shown in Figure 5.1. The static semantics of *Tiny* require that all identifiers be declared, that a single identifier may not be defined twice in the same block (although a block may have the same name as a variable declared in that block), and that the operands of an expression must be either INTEGER or REAL. The visibility rules specify nested block structure: references to identifiers are resolved by searching sequentially through the enclosing scopes until a declaration for that identifier is bound, or until no declaration can be found.

Appendix B provides complete LADLE and COLANDER descriptions of *Tiny*. Excerpts from the COLANDER description of *Tiny* will be used throughout this chapter and Chapter 6 to illustrate points of discussion.

```
⟨program⟩    = "PROGRAM" ⟨block⟩
⟨block⟩      = id "{" ⟨decls⟩ ";" ⟨stmts⟩ "}"
⟨decls⟩      = ⟨declaration⟩ + ";"
⟨declaration⟩ = ⟨idlist⟩ ":" id
⟨stmts⟩      = ⟨statement⟩ + ";"
⟨idlist⟩     = id + ","
⟨statement⟩  = ⟨assign⟩ | ⟨block⟩
⟨assign⟩     = id ":=" ⟨expression⟩
⟨expression⟩ = ⟨expression⟩ "+" ⟨expression⟩
             | ⟨expression⟩ "*" ⟨expression⟩
             | "(" ⟨expression⟩ ")"
             | id
```

Figure 5.1: Syntax of *Tiny*

5.1 Executing a Colander Description

An expression in the target language is represented in the computer using a syntax tree. The syntax tree encodes the syntactic structure of the expression; contextual constraints are expressed using goal clauses associated with structures in the tree. The syntax tree is used in *Pan* for incremental parsing, for detection and recovery from syntactic errors, for structure-oriented editing, and for structure-oriented viewing (pretty-printing).

Evaluating a LCG consists of visiting subtrees and attempting the goals associated with those subtrees. In general a goal can use rules, create data, allocate collections, and generate values for properties. When a goal is attempted, the subtree associated with the goal is said to "own" any collections or values created by that goal.

In a LCG, each subtree in the syntax tree is assigned a distinguished datapool called a *context datapool*. Most of the goals associated with a syntactic structure depend on values in the context datapool. By default in COLANDER, the subtree's context datapool is automatically propagated to its children. Goals associated with a subtree are not attempted until that subtree receives a context datapool.

COLANDER partitions the goals associated with a syntactic structure into two classes: those whose primary use is to establish the evaluation context used by that structure or by its substructures, and those goals whose primary use is to express a static-semantic constraint of the target language. Goals in the first class are called **first-pass** goals. They are evaluated during a top-down preorder walk of the syntax tree. Goals in the second class are called **second-pass** goals. They are evaluated after the first-pass goals, and in the current implementation, are evaluated after the first-pass goals of the subtree's children have been evaluated.

Initially, the COLANDER evaluator traverses the syntax tree propagating context datapools by evaluating first-pass goals. By virtue of the node-reuse heuristic[1], every new or

[1]Recall that after incremental parsing, every node present in the syntax tree is categorized as "new", "reused", or "unchanged" (Section 2.7.2).

reused tree node appears on a path, starting at the root of the syntax tree, that contains only new or reused nodes. During the first phase of evaluation, the first-pass goals are attempted for all nodes that are new, or that are reused, or that are unchanged but have received a new context. Generally, first-pass goals establish the contextual information propagated to child subtrees, but other actions can be taken as well. This traversal can cease once it reaches an unchanged node that receives the same context datapool as it previously received. Following the first pass, unsatisfied goals are repeatedly attempted until no further progress can be made in satisfying goals.

Note that even though a node in the syntax tree may be unchanged and may receive the same context datapool, the contents of the context datapool may have changed. The consistency manager handles this case automatically.

The incremental LCG evaluator in *Pan* invokes the COLANDER language interpreter to attempt a goal.

5.2 The COLANDER Interpreter

The COLANDER interpreter is a structure-sharing, continuation-passing, extensible interpreter modeled on the Foolog interpreter [97, 98]. It interprets a simple intermediate form that results from compiling a COLANDER description.

COLANDER compiles source clauses into an intermediate form amenable to efficient interpretation. The compiler has two passes. During the first pass, references to subtrees are converted into calls to internal functions, variables are assigned indices into binding environments, and a number of syntactic forms are normalized. During the second pass, procedure names are replaced with integer indices into definition tables and many of the primitive language forms are transformed into alternate internal representations. For example, literals that represent facts are converted into calls to the `get-fact` internal function.

The COLANDER interpreter works closely with the consistency manager, as discussed above and in Chapter 6.

5.3 Procedures, Goals, and Data

The clauses of a COLANDER description are partitioned into three categories: procedures clauses, goal clauses, and data literals.

5.3.1 Procedures

A set of program and unit clauses whose heads have the same functor name defines a **procedure. Procedure clauses** represent one or more alternatives in the definition of a procedure by cases. The first clause of the definition for append given by

```
append([],X,X).
append([A|B],C,[A|D]) :- append(B,C,D).
```

is considered a procedure clause in this classification, whereas in PROLOG it would be called a "fact."

⟨*goal*⟩ = ":-" ⟨*goal_clause*⟩ "."
⟨*goal_clause*⟩ = ⟨*goal_functor_list*⟩ + "|"
⟨*goal_functor_list*⟩ = ⟨*functor_list*⟩

Figure 5.2: Syntax of Goals

Unlike PROLOG, the current implementation of COLANDER does not allow procedure clauses to be dynamically added during the evaluation of a logical constraint grammar. This restriction is discussed in Section 6.6.1.

5.3.2 Goals

Goals can have multiple alternative clauses, each alternative separated by a vertical bar. The alternatives are evaluated in their order of appearance. A goal is satisfied whenever one of its alternatives is satisfied.

The syntax for a goal is shown in Figure 5.2. For example, the following goal in *Tiny* checks that an identifier has been declared[2].

```
:- pname($id, ?name),
   context(?context),
   <scope(?scope), ?context>,
   <lookup(bound(?name,?entity)), ?scope>:
                          ("~a is not declared! ~%", ?name).
```

It first determines the name associated with the identifier node and finds the collection that represents the current scope. It then invokes "lookup" to search for a binding of the identifier. If no binding is found, the invocation of "lookup" generates an error message. The details are explained in subsequent sections.

Goals independent of any syntactic structure are evaluated once, in order of their appearance, prior to attempting the goals associated with subtrees in the abstract syntax tree. This gives the writer of a language description the opportunity to embed "executable" code into the description in order to allocate entities or perform other initializing actions. If such a goal fails, it is impossible for the consistency manager to achieve consistency.

Goals associated with subtrees of the syntax tree are either satisfied or unsatisfied; the consistency manager is always attempting to satisfy unsatisfied goals.

5.3.3 Data Literals and Data Values

COLANDER provides two kinds of data literals: **facts** and **properties**. Data literals are represented by unit clauses that are not procedure clauses.

[2]Logical variables are denoted by a prefixed *?*; the construct <*?term*, *?collection*> indicates evaluation of the given literal *?term* relative to the specified collection.

COLANDER supports values of type integer, real, symbol, string, list, and "structure", together with the various types of collections—datapools, entities, and subtrees. (A collection may be treated like a value.) A **structure** is just a functor.

A fact is a COLANDER functor having a name and any number of arguments. For instance, a fact representing the binding of a name to an entity might be represented by the functor "bound(*?name*, *?entity*)" where the variables *?name* and *?entity* must be bound to ground values when the fact is added to a datapool. The symbol "bound" is the name of the fact, while the other two fields are called the arguments of the fact. Facts reside in datapools.

Subtrees and entities are collections that can have properties. A **property** is a named value associated with a collection. Properties are single-valued. Thus, it is not possible for a collection to have more than one property value with a given property name, making access to the values of properties quite efficient. A property is denoted by a functor of two arguments:

property-name(*entity-or-subtree*, *value*)

The name of the functor is the name of the property, the first argument denotes the collection, and the second argument is the value. For example, the literal "pname($id, *?name*)" denotes a subtree property. The name of the property is "pname", the subtree is denoted by "$id", and the value of the property will be matched to the logical variable "*?name*". The "pname" property accesses the character string that is the external representation (print name) of an identifier node.

5.4 Datapools, and Entities

A datapool is a repository for facts. Datapools are used to represent context datapools and may also be used for other purposes. Entities are collections that can be allocated and manipulated by the description writer. Entities can be used to represent linguistic objects of the document being analyzed. Subtrees are structures in the syntax tree that represents a document. Subtrees are discussed in detail in the next section.

Multiple datapools require that the assert and retract facilities of PROLOG be extended to assert and retract relative to a specified datapool. However, in general, COLANDER facts are not explicitly retracted. The consistency maintenance aspect of the COLANDER evaluator handles the task of retraction.

Datapools and entities are explicitly allocated and manipulated. They are owned by the subtree in the syntax tree at which they are allocated. Subtrees are created and destroyed during syntax analysis. They can be annotated using subtree properties.

Datapools and entities are persistent. Once created, datapools and entities are permanently associated with their owning subtree. Collections are destroyed only when their owning subtree is removed from the syntax tree.

5.4.1 Datapools and Context Datapools

Datapools contain facts. Facts are added to datapools by the primitive function assert. Facts are usually removed from datapools by the consistency maintenance process;

removal of facts is largely beyond the control of the writer of a description.

A **context datapool** is a distinguished datapool that is passed from subtrees to their children. In COLANDER, each subtree of the syntax tree receives a single context datapool from its parent. By default, the context of a subtree is passed directly to its children. If a subtree does not have a defined context, none of its goals will be attempted. The root of the entire syntax tree automatically receives the global datapool as its initial context datapool.

At any subtree in the syntax tree, the primitive function context(*?x*) binds the context datapool of that subtree to the variable *?x*. This primitive fails whenever a context datapool has not yet been assigned to that subtree.

5.4.2 Evaluation Relative to Datapools

The definition of a LCG description requires that multiple datapools be supported. In COLANDER, the notation "<*expr*, *datapool*>" denotes the evaluation of the expression *expr* relative to the datapool *datapool*. This construct uses only the specified datapool. It succeeds or fails based on the contents of that datapool.

Connections between datapools may be specified by meta-level evaluation clauses. For example, a simple COLANDER search rule for block-structured languages is given by the following procedure from *Tiny*.

```
<lookup(?term), nil> :- !, fail().
<lookup(?term), ?pool> :- <?term, ?pool>.
<lookup(?term), ?pool> :- <parent(?parent), ?pool>,
                          <lookup(?term), ?parent>.
```

The search rule operates by attempting to evaluate a literal "*?term*" relative to a given collection of facts "'*?pool*". It contains three program clauses. The first clause indicates that if there are no further collections to be searched, the search should fail. The second clause attempts to evaluate the literal term relative to the given collection. Finally, the third clause determines the next collection to be checked, and recursively invokes "lookup" to evaluate "*?term*" with respect to that collection. The next collection to be searched is specified by a fact "parent(*?parent*)" that is located in the collection presently being searched. The fact named "`parent`" provides access to the datapool that represents the enclosing scope for the scope represented by "*?pool*".

5.4.3 Entities

Entities are collections that can be used to represent objects in the language being defined. Information about entities is represented using named values called **entity properties**. An entity property is a named value associated with an entity. They are more space-efficient than datapools, but cannot be used to store facts *per se*.

For example, an entity might be used to represent a variable in a program; the type of that entity might be represented by a string-valued entity property called `mytype`. Then the following goal would create a new entity and define its type to be `"integer"`:

:- **new-entity**(*entity-name, ?entity*), **assert**(**mytype**(*?entity,* "integer")).

In practice, the value of **mytype** would be more complex than just a character string; it could even be another entity. Bindings between entities and names can be established using facts.

Properties of an entity can be added at any time. Values of properties are usually removed by the consistency maintenance process, either when the value is changed or when the assertion of the value is no longer in effect.

5.5 Subtrees

To facilitate communication between subtrees in the syntax tree, COLANDER provides subtree attribution. The mechanism provided by COLANDER is considerably more restricted than that provided by attribute grammars. In return, COLANDER provides a more encompassing environment for describing and enforcing contextual constraints.

A subtree represents a structure in the syntax. Subtrees are managed by the incremental parser; currently they are not directly created or destroyed by actions in a COLANDER description.

The notation $*subtree-name* or $*subtree-name*<*index*> is used to designate the various subtrees of a structure in the syntax tree. *Subtree-name* must be the name of a symbol appearing in the production. If a symbolic name appears more than once in the production, the optional integer *index* must be used to distinguish the occurrences. The occurrence of a name on the left-hand side of the production is given index 0. Occurrences on the right-hand side of the production are indexed from 1, with the indices increasing from left to right.

Subtree properties are named values associated with subtrees of the internal tree. They are divided into three subclasses: structural, maintained, and extrinsic.

5.5.1 Structural Properties

The value of a **structural** property depends on the immediate structure of that subtree. Structural properties are maintained by the consistency manager. They are defined using special facilities. A number of structural properties are available. The table of Figure 5.3 presents some of the structural properties provided by COLANDER.

Structural properties are classified into two categories: those whose values can change if the subtree was split and reused during parsing, and those whose values can change if their parent was split and reused during parsing. The consistency manager uses this classification to determine which structural properties to re-attempt. The goals that depend on a structural property are re-attempted only when the value of the property changes.

5.5.2 Maintained Properties

The value of a **maintained property** is defined by a local procedure associated with that subtree. Syntactically, such procedures resemble subtree-local procedures whose

childN(*?node*,*?value*) — Binds *?value* to the N^{th} child where $0 \leq N \leq 9$

children(*?node*,*?value*) — Binds *?value* to list of all children of *?node*

current-node(*?value*) — Binds *?value* to current node

in-tail-of-sequence?(*?node*) — Succeeds if *?node* is not the first child in a sequence

is-leftmost-child(*?node*) — Succeeds if *?node* is the leftmost child of parent

is-first-in-sequence?(*?node*) — Succeeds if *?node* is the first child in a sequence

is-last-in-sequence?(*?node*) — Succeeds if *?node* is the last child in a sequence

is-rightmost-child(*?node*) — Succeeds if *?node* is the rightmost child of parent

left-sibling(*?node*,*?value*) — Binds *?value* to the subtree to the left of *?node*

pname(*?node*,*?value*) — Binds *?value* to the textual name of a leaf node

right-sibling(*?node*,*?value*) — Binds *?value* to the the subtree to the right of *?node*

Figure 5.3: Structural Properties Provided by COLANDER

names are the same as maintained properties. In a sense, maintained properties play the role of attributes in an attributed grammar, and the procedures that define the value of the property play the role of attribute functions.

The value of a maintained property is computed relative to the subtree for which the property is being defined. Naturally, the values of maintained properties can depend on any other values available to a local procedure, including the properties of the parent, child, or immediate sibling subtrees.

Maintained properties generally encode local information such as the type of an expression. They differ from facts by being properties of a subtree and by being defined by a procedure. The syntactic form of a maintained property is: *name*(*subtree-reference*, *?value*) where *subtree-reference* refers to a neighboring subtree in the tree.

For instance, suppose that subtrees representing expressions have a maintained property for the type of the expression. Consider the portion of *Tiny* that type-checks expressions involving the addition operator shown in Figure 5.4. It defines two maintained properties. "Num-operands" is a maintained property whose value is the number of operands in the subexpression. "Expr-type" is a maintained property whose value is the resulting type of the expression. In *Tiny*, the procedure `result-type` given by

```
result-type("integer", "integer", "integer").
result-type("integer", "float", "float").
result-type("float", "float", "float").
result-type("float", "integer", "integer").
```

is a relation that defines the type-conversion rules for the language.

In Figure 5.4, the "expression" subtree on the left-hand side of the production is denoted `$expression<0>`, while the rightmost subtree is denoted by `$expression<2>`.

```
expression = expression "+" expression => expr_plus {
             num-operands($expr_plus, ?nop) :-
                                              num-operands($expression<1>, ?n1),
                                              num-operands($expression<2>, ?n2),
                                              +(?n1, ?n2, ?temp),
                                              +(?temp, 1, ?nop).

             expr-type($expr_plus, ?type) :-
                                            expr-type($expression<1>, ?t1),
                                            expr-type($expression<2>, ?t2),
                                            result-type(?t1, ?t2, ?type).
}
```

Figure 5.4: Type-checking in *Tiny*

Note that the value of **expr-type** at the current node depends on the values of **expr-type** defined for the children of the current node.

The flow of property values in COLANDER trees is usually much simpler than in attribute grammars. Usually, inherited values should be propagated using the context datapool, and most values will reside in the database. While it is possible to directly emulate an attribute grammar, it is far more efficient to use the datapools and the context for moving values through the tree.

This restriction is not as severe as it might appear. In most attribute grammar descriptions, inherited attributes either summarize relatively local structural information about the program tree or else they consist of a "symbol table" containing non-local information. COLANDER subsumes the "symbol table" into the database along with other information about the expression. Local structural information can be passed like inherited attributes by creating and propagating new context datapools containing that information.

Synthesized values (those moving from the leaves towards the root of a subtree) are still passed as in attribute grammars. Such properties appear frequently in a COLANDER description.

In practice, there is no need to evaluate a maintained property function until its value is needed to satisfy a goal. COLANDER retains the values computed by maintained property functions in order to minimize recomputation.

5.5.3 Extrinsic Properties

Agents other than the COLANDER evaluator can add and remove properties from a subtree. An **extrinsic property** is a subtree property that is neither a structural property nor a maintained property. Extrinsic properties are usually manipulated by programs or components via a client interface, although they can appear in an LCG description. Extrinsic properties are not under consistency maintenance unless they are declared and used within the language description.

```
⟨iterator⟩ = ⟨iter_name⟩ "(" ⟨op_or_list_kw⟩ "," /*operator*/
                             variable "," /*child variable*/
                             ⟨vbl_or_rail⟩ "," /* list of children */
                             ⟨iter_clauses⟩ ","
                             ⟨vbl_or_rail⟩ "," /* result vars */
                             variable /* final result */
                             ")"

⟨op_or_list_kw⟩ = ⟨alpha_id⟩
                | ":list"
⟨iter_name⟩ = "for-all-children"
            | "for-each-child"
⟨vbl_or_rail⟩ = variable
              | ⟨rail⟩
⟨iter_clauses⟩ = "{" ⟨iter_clause_seq⟩ "}"
⟨iter_clause_seq⟩ = ⟨iter_clause⟩ + "||"
⟨iter_clause⟩ = ⟨iter_kw⟩ "=>" ⟨goal_clause⟩
⟨iter_kw ⟩ = ":all"
           | ":all-but-first"
           | ":all-but-last"
           | ":middle"
           | ":first"
           | ":last"
           | ":rest"
           | ":singleton"
```

Figure 5.5: Syntax of Mapping Functions

5.5.4 Mapping Goals over Sequence Subtrees

The syntax tree used in *Pan* allows subtrees with an arbitrary number of children called *sequence nodes*. All of the children of a sequence node have the same operator. The children of a sequence subtree can be obtained by using the structural property function children(*subtree-ref, ?child-list*). COLANDER provides two functions for mapping goals over the children of a sequence subtree.

The first, for-each-child, applies a given iterator goal to each child in a specified list of children. It succeeds if any of the goal succeeds at any of the children. The second, for-all-children, is like for-each-child, but succeeds only if the goals succeed for all of the specified children. In both for-each-child and for-all-children, the operator of the child subtrees must be specified. If the operator is the keyword `:list`, the mapping function is taken to be a list operator. Figure 5.5 shows the syntax for a mapping function.

A call to for-each-child is specified by

`:all`	Selects all children in *?child-list*
`:all-but-first`	Synonym for `:rest`
`:all-but-last`	Selects all children except the last in *?child-list*
`:first`	Selects the first child in *?child-list*
`:last`	Selects the last child in *?child-list*
`:middle`	Selects all children that are neither first nor last in *?child-list*
`:rest`	Selects all children except the first in *?child-list*
`:singleton`	Selects the only child in a single-element *?child-list*

Table 5.1: Selector Keywords for Mapping Functions

---

for-each-childoperator, *?child*, *?child-list*, *iter_clauses*, *?result-vars*, *?result*)

The mapping function binds *?child* to each subtree in *?child-list* (in left-to-right order) and evaluates the terms in *iter_clauses*. Following each successful evaluation of *iter_clauses*, the values of the variables in *?result-vars* are saved; following the execution of for-each-child, the variable *?result* is bound to a list of values representing the values of *?result-vars* after each successful visit to a child. If *?result-vars* is a single variable, then *?result* will be a list of values; if *?result-vars* is a list of variables, then *?result* will be a list of lists. The call to for-each-child succeeds if any application of the iterator goals to a child succeeds.

In contrast to for-each-child, the mapping function for-all-children succeeds if and only if the given iterator goal succeeds at every specified child. As with for-each-child, the *?result* variable will be bound to a list of values if the iterator succeeds. A call to for-all-children is identical (except for the name) to for-each-child.

For both mapping functions, the value of the variable *?child-list* specifies the children to be visited. Children are be visited in the order that they appear in *?child-list*. A iterator goal is selected relative to the order specified in *?child-list*. It is an error if *?child-list* is not bound when the iteration form is executed.

The ⟨*iter_kw*⟩ field determines which ⟨*goal_clause*⟩ of the *iter_clause* is applicable. Each *iter_clause* is examined in order. The ⟨*goal_clause*⟩ for the first ⟨*iter_kw*⟩ field that applies is attempted. Elements in the ⟨*iter_kw*⟩ field are described in Table 5.1.

In the description of *Tiny*, a mapping function is used to declare all of the identifiers that appear within a declaration. The COLANDER code for declaring new variables is shown in Figure 5.6. A mapping function "for-all-children" is used to create a declaration for each child of "$idlist". The single goal first binds variables to hold the name of the type identifier ("*?type-name*"), the current context ("*?context*"), the current scope ("*?scope*"), the name of the current scope ("*?block*") and the children of the "$idlist" node ("*?children*"). The mapping function is then used to iterate over all of the elements of "*?children*". For every child in "*?children*", the iterator goal accesses the character string associated with the child and checks whether a binding for that name is present in the current scope. If no such binding is present, the goal proceeds to create a new entity that will represent the

```
declaration = idlist ":" id => op_decl {
              //
              //
              :- pname($id, ?type-name),
                 context(?context),
                 <scope(?scope), ?context>,
                 <blockname(?block), ?scope>,
                 children($idlist, ?children),
                 for-all-children(id, ?child, ?children,
                 { :all => pname(?child, ?name),
                           not(<bound(?name, ??), ?scope>):
                            ("~a is already bound in scope ~a!~%", ?name, ?block),
                           new-entity(binding, ?ent),
                           assert(bound(?name, ?ent), ?scope),
                           assert(type(?ent, ?type-name))
                 }, ?name, ?namelist).
              }
```

Figure 5.6: Declarations in *Tiny*

variable being declared. Finally, the iterator goal creates a new fact "bound" and a new entity property "type" and asserts them in the appropriate collections.

5.6 Operators

Each structure in the syntax tree is described by an **operator**. Operators are defined as part of the abstract grammar in a LADLE description. They correspond to productions in the context-free grammar underlying the LCG. Semantic definitions and goals are attached to an operator by extending the LADLE definition of the operator. The syntax for an operator appears in Figure 5.7. Figure 5.6 shows a single operator named "op_decl", while Figure 5.8 shows the full description for the operator "op_block".

An operator can have local procedures and goals associated with it. The goals associated with an operator are categorized into first- or second-pass goals, depending on when those goals are to be attempted.

The nonterminal ⟨*rhs*⟩ names the subtrees in the syntax tree. If the operator does not denote a sequence node in the tree, the children of the operator can be individually named using a subtree reference.

Operator descriptions contain three optional parts: local definitions of collections and procedures, goals that must be satisfied during propagation of context datapools, and goals that must be satisfied during normal evaluation.

```
⟨operator⟩ = ⟨alpha_id⟩ "=" ⟨operator_def_seq⟩
           | ⟨alpha_id⟩ "=>" "{" ⟨op_defs⟩ "}"
⟨operator_def_seq⟩ = ⟨operator_rhs_definition⟩ + "|"
⟨operator_rhs_definition⟩ = ⟨prod_rhs⟩ "=>" "{" ⟨op_defs⟩ "}"
                          | ⟨prod_rhs⟩ "=>" ⟨alpha_id⟩ "{" ⟨op_defs⟩ "}"
⟨op_defs⟩ = ⟨local_decls_part⟩
          | ⟨local_decls_part⟩ "//" ⟨first_pass_goals⟩
          | ⟨local_decls_part⟩ "//" ⟨first_pass_goals⟩ "//" ⟨second_pass_goals⟩
⟨prod_rhs⟩ = ⟨rhs⟩
⟨rhs⟩ = ⟨grammar_symbol_seq⟩
      | ⟨grammar_symbol_seq⟩ "+" ⟨opt_grammar_symbol⟩
      | ⟨grammar_symbol_seq⟩ "++" ⟨opt_grammar_symbol⟩
      | ⟨grammar_symbol_seq⟩ "*" ⟨opt_grammar_symbol⟩
      | "[" ⟨grammar_symbol_seq⟩ "]"
⟨grammar_symbol_seq⟩ = ⟨grammar_symbol⟩*
⟨grammar_symbol⟩ = ⟨alpha_id⟩
                 | string
⟨opt_grammar_symbol⟩ = [ ⟨grammar_symbol⟩ ]
⟨local_decls_part⟩ = ⟨local_decl⟩*
⟨local_decl⟩ = ⟨entity_name_decl⟩
             | ⟨datapool_name_decl⟩
             | ⟨context_name_decl⟩
             | ⟨rule⟩
⟨first_pass_goals⟩ = ⟨goal⟩*
⟨second_pass_goals⟩ = ⟨goal⟩*
```

Figure 5.7: Syntax for Operators

Local Declarations

Each collection owned by a subtree representing this operator must be named. The names are declared in the local declarations portion of the operator description. Local declarations include definitions of datapools, context datapools, entities, local procedures, and maintained property procedures. The operator for a "block" in the description of *Tiny* shown in Figure 5.8 . The operator has two local definitions: one for a new context and one for a new datapool.

Local Procedures

Local procedures are defined within the operator description where they are used. Their definition is visible only to other procedures and goals in that operator description.

The computation rules for maintained properties resemble procedure definitions

```
block = id BEGIN decls stmts END => op_block {
        context-name(mycon).
        datapool-name(local-scope).
        //
        :- new-context*(mycon, ?newcon),
          context(?context),
          <scope(?parent), ?context>,
          new-datapool(local-scope, ?scope),
          assert(scope(?scope), ?newcon),
          assert(parent(?parent), ?scope),
          pname($id, ?blockname),
          assert(blockname(?blockname), ?scope).
        //
        :- context(?context),
          <scope(?outer), ?context>,
          pname($id, ?name),
          not(<contains-block(?name), ?outer>) :
              "Illegal to declare a block twice in the same scope! ~%",
          assert(contains-block(?name), ?outer).
}
```

Figure 5.8: Operator for Blocks in *Tiny*

within the clauses associated with an operator. The head of a maintained property procedure will always be identical to a named property function: *name*(*subtree-ref*, *?value*) where the first argument is a subtree reference or a variable.

Simple local procedures differ from property-defining procedures in that their values are not retained and are not available outside of that subtree. Figure 5.4 (page 95) shows an operator that defines two maintained property procedures, but contains no other declarations or goals.

First-Pass Goals

The first-pass goals of the operator description usually include the goals that allocate and propagate context datapools. It is guaranteed that first-pass goals will be attempted prior to attempting the second-pass goals. Since first-pass goals are evaluated before the first-pass goals of the subtrees of the operator, they should not depend on properties of the subtrees. The operator for a block shown in Figure 5.8 has a single first-pass goal that allocates the new context and the new datapool that are inherited by the declarations and statements within the block.

fact(*name([arg1, ... , argN])*, [*documentation*])
entity-property(*name*, [*documentation*])
maintained-property(*name*, [*documentation*])
node-property(*name*, [*documentation*])
context-name(*name*, [*documentation*])
datapool-name(*name*, [*documentation*])
entity-name(*name*, [*documentation*])

Figure 5.9: Primitive Functions for Declaring Facts, Properties, and Collections

Second-Pass Goals

Second-pass goals are attempted after the first pass goals, and possibly after the subtrees of the operator have been visited. Second-pass goals usually describe the contextual constraints on the structure described by that operator. Figure 5.8 shows a second-pass goal that ensures that a block name is not declared more than once in the same scope.

5.7 Declarations

In a COLANDER description, the names of all collections, facts, and properties must be declared before they are used. Facts and properties must be declared globally, outside any operator definition. Collections may be declared either globally or locally within the definition for an operator. The declarations for facts and properties appearing in the description of *Tiny* are given by

```
fact(bound(?name, ?entity), "Represents a binding of a name to a variable.").
fact(blockname(?type-name), "Holds the name of the current block.").
fact(parent(?parent), "Parent of current scope.").
fact(scope(?parent), "Current scope in current context.").
fact(contains-block(?name), "Represents the declaration of a block.").

entity-property(type, "Represents the type of an entity.").

maintained-property(expr-type, "Synthesized type of expression tree").
maintained-property(num-ops, "Number of operands in expression").
```

5.7.1 Declaring Facts and Properties

Properties have fixed arguments; there is no need to declare an argument list for them. Facts do not have fixed argument lists; so their declarations must include the argument list. Figure 5.9 illustrates the primitive forms used to declare facts, properties, and collections.

5.7.2 Declaring Collections

Every collection owned by a subtree in the syntax tree has a name. The names must be declared within the description for that subtree, using one of the functions **datapool-name**, **context-name**, or **entity-name**. These functions take two arguments: the symbol being declared and an optional documentation string:

name-declarator(*symbol* [, *documentation*])

where *name-declarator* is one of **context-name**, **datapool-name**, or **entity-name**.

5.7.3 Allocating Collections

Once the names have been declared, the allocators **new-datapool**, **new-context**, **new-context***, and **new-entity** can be used to create the collection. The name of the collection must be specified when it is allocated. After allocation, the collection can be accessed by using the functor whose name is the name of the collection and whose argument is a variable that will be bound to that collection. Examples from *Tiny* can be found in Figure 5.8.

For example, the following clauses are legal within the description of an operator:

```
/* Declare a name */
datapool-name(LOCAL-DP, "locally allocated pool").
//
/* Allocate pool */
:- ..., new-datapool(LOCAL-DP,?dp), ...
/* Now access the pool in another clause */
:- ..., LOCAL-DP(?new-dp), ...
```

To assert facts into a datapool at allocation time, the allocators for datapools accept an optional sequence of facts. The allocator for entities, **new-entity** supports a similar protocol, except that property values rather than facts are created.

The simplest case is that of **new-datapool**:

new-datapool(*symbol, ?new-pool* [, *list-of-facts*]).

new-datapool has 3 arguments: the name of the datapool being created, a variable that will be bound to the result of the allocation, and a sequence of facts. To create a new datapool containing the fact `record-pool()`, one would use

new-datapool(*symbol, ?dp, (* `record-pool()` *)*).

The functions **new-context** and **new-context*** create and propagate context datapools. The function **new-context** requires that the children that will receive the new context datapool be specified explicitly, while **new-context*** propagates the new context datapool to all of the children. If a node does not create a new context datapool, the context datapool that it received is propagated to all of its children.

new-context(*symbol, ?env, list-of-kids* [, *facts*])
new-context*(*symbol, ?env* [, *facts*])

new-context is a function of 4 arguments: the datapool name, the result, a list of children that will receive the context, and (optionally) the facts to assert. **new-context*** is a shorthand for the case of **new-context** where the new context datapool is propagated to all of the subtree's children.

The allocator **new-entity** parallels **new-datapool**:

new-entity(*symbol, ?new-pool* [, *properties*])
new-anon-entity(*?new-pool* [, *properties*])

Values asserted by **new-entity** must be entity property values. **new-anon-entity** can be used only in goals that are independent of any syntactic structure. It creates and returns a new (unnamed) entity.

5.8 Assert, Retract, and Consistency Maintenance

Facts and properties are asserted as goals are satisfied. Values remain in the database until the goals that asserted them are re-evaluated and generate new values, or until the subtree that owns them is removed from the syntax tree.

The sole datapool for which assertion and removal of facts is controlled by the description writer is a special datapool in which all "global" definitions reside. The global pool is *not* maintained by the consistency maintenance process. The global datapool can be accessed using the primitive function **global-datapool**(*?dp*).

The **assert** family of functions add facts and properties to the database. Usually, the consistency maintenance component of COLANDER is responsible for removing information from the database. However, **retract** function may be used to explicitly remove a fact from the global database, or to remove properties via the client interface to the consistency manager.

There are two forms of **assert**:

assert(*value*)
assert(*value, collection*)

The two-argument form, specifying an explicit collection, is more common. When this form is evaluated, the *value* is added to the specified collection. The single argument form of **assert** adds a value to the global datapool.

In both cases *value* must be a fact or a property or a variable bound to a fact or property, and may not contain any unbound variables.

Like **assert**, there are two forms for **retract**:

retract(*value*)
retract(*value, collection*)

The function **retract**(*value, ?col*) can be used to remove a value from the collection specified by *?col*. The function **retract**(*value*) can be used to remove a value from the global datapool.

5.9 Database Triggers

COLANDER provides database triggers called *on-conditions*. The trigger is activated when values are added or removed from the database. Each trigger has the form

ON ⟨*condition*⟩ ⟨*data-literal*⟩ ⟨*clauses*⟩

where ⟨*condition*⟩ is either "**assert**" or "**retract**". When a data value $\mathcal{D}$ that matches ⟨*data-literal*⟩ is asserted (retracted), then $\mathcal{D}$ is unified with ⟨*data-literal*⟩ to create a substitution θ, and then the goal "$\leftarrow$ (⟨*clauses*⟩)θ" is attempted. All solutions to the goal are calculated.

5.10 Errors and Error Messages

Any literal appearing in a goal or procedure body can be suffixed with an error message. If a goal fails during evaluation, the error message associated with the most recent literal to fail is captured and retained for later use. Thus, the error messages created during backtracking are usually not retained. The syntax for error messages is given by

⟨*functor*⟩ = ⟨*rule_head*⟩
| ⟨*rule_head*⟩ ":" ⟨*message*⟩
⟨*message*⟩ = string
| "(" string ")"
| "(" string "," ⟨*message_arg_seq*⟩ ")"
⟨*message_arg_seq*⟩ = ⟨*literal*⟩ + ","

Error messages are separated from a rule head (functor) by a colon (":"). The string is interpreted as a Common Lisp format string [65].

5.11 Special Functions Used In Conjunction with Consistency Maintenance

COLANDER provides two special functions that interact with the consistency manager. The function all-solutions can be used to gather up all solutions to a goal. It is re-evaluated when the set of solutions might have changed. The function notever is a form of negation that is monitored by the consistency manager.

5.11.1 Bagof and All-Solutions

COLANDER supplies two ways to gather all of the results from attempting a goal: bagof and all-solutions.

bagof(*result-vars, goal, ?result*)
all-solutions(*result-vars, goal, ?result*)

bagof is the normal PROLOG "bagof" operator. For each solution of *goal*, the values of the variables in *result-vars* are gathered into *?result*. If *result-vars* consists of a single variable, then *?result* will be a list of values; if *result-vars* is a list of variables, then *?result* will be a list of lists of values. All solutions of *goal* are calculated. No ordering of the values in *?result* is implied. Duplicate values may appear in *?result*.

Unlike **bagof**, the results of the **all-solutions** primitive are under consistency maintenance. Thus if a new solution could arise, the goal containing the **all-solutions** will be retried. The function **all-solutions** is semantically equivalent to **bagof**, except that the evaluation of **all-solutions** causes the results to be retained for use by the consistency manager.

For instance, attempting

:- bagof(*?X*, <bound("name",*?X*), *?scope*>, *?result*)

gathers into *?result* all of the values *?X* for which bound("name",*?X*) is defined in the datapool *?scope*. However, the consistency maintenance component of COLANDER may not detect that this goal should be retried when the fact bound("name", abc) is later added to the datapool *?scope*. On the other hand, the goal

:- all-solutions(*?X*, <bound("name",*?X*), *?scope*>, *?result*)

behaves just like **bagof** in the above case, but also creates dependency information so that if a new solution becomes available, the goal will be retried. One use for **all-solutions** is for gathering together all of the definitions of a name in order to resolve overloading in Ada programs.

5.11.2 Not and Notever

COLANDER provides two ways to describe negation: **not** and **notever**.

not(*goal*)
notever(*goal*)

not is the normal PROLOG "not" operator. It is implemented using the negation as failure rule [94]. **not**(*goal*) succeeds only when the *goal* fails. If the negation succeeds, none of the variables bound within the **not** retain their bindings. The **not** primitive is not under consistency maintenance, so that it will not be re-evaluated just because of a change that might make *goal* succeed. This means that a goal can use **not** and then assert a value that would make the **not** fail within the same action without causing infinite regress. Goals like

:- not(bound("name", "object")) ... assert(bound("name", "object"))

are perfectly satisfactory and necessary.

Unlike **not**, the results of the **notever** primitive are under consistency maintenance. If a new solution arises that would cause the **not** to fail, then the goal containing the **notever** will get retried. The evaluation of **notever** is semantically equivalent to **not**, except that the evaluation of **notever** causes the results to be stored for use by the consistency manager.

For instance, attempting

:- not(<bound("name",*?X*), *?scope*>)

when bound("name",*?X*) is not present in the datapool *?scope* will succeed. This goal may not be retried when a fact of the form bound("name", abc) is added to the datapool *?scope* at a later time. On the other hand, the goal

:- notever(<bound("name",*?X*), *?scope*>)

will succeed so long as any fact of the form bound("name",*?X*) is not present in *?scope*. If the fact bound("name", abc) is later added to the datapool *?scope*, this goal will be retried.

5.11.3 Access to LISP

COLANDER provides easy access to LISP facilities by using the lisp function:

lisp(*lisp-expr*, *?result*)

The primitive lisp evaluates its first argument and binds the result to the second argument.

5.12 Peeking Into the Implementation

The COLANDER interpreter is itself extensible on two levels: one may add new COLANDER rules and definitions at the "source-code" level, and one may add new "builtin" definitions that will improve execution performance. New structural properties can also be defined.

COLANDER treats interpreter primitives specially in two ways. First, they may be implemented by LISP functions that have access to the internal data structures of the interpreter. Second, even if they are implemented using normal COLANDER procedures, primitives get special treatment during the analysis of shadowing.

A simplified interface has been provided that allows one to define relatively simple and efficient LISP functions that do not require complicated access to the runtime state of the interpreter. Such functions are categorized as either predicates or procedures; predicates return a boolean result, where procedures return a calculated value.

Adding functions that do require access to the state of the interpreter is much more complicated, and will require the advice of an expert.

5.13 Simple Access to LISP

A simple LISP primitive is a predicate or a procedure. The implementing form for a predicate returns a boolean value. The definition of a predicate describes both the COLANDER arguments that are passed to the implementor together with a LISP form that will be evaluated when the primitive is invoked. For instance, the definition of the string equality predicate, string=, is given by

```
(define-lisp-predicate string= (?x ?y) (and (stringp ?x) (stringp ?y)
                                            (string= ?x ?y)))
```

The first argument to **define-lisp-predicate** is the name of the new predicate; the second argument describes the input variables, and the third argument is the LISP form that implements the predicate. When the form is actually evaluated, the variables in the form that appear in the argument list will be bound to the appropriate values. Since this is a predicate, no result variable is necessary.

A procedure, on the other hand, has both input variables and a result variable. For example, the definition of the numeric plus operation is given by

```
(define-lisp-procedure + ((?x ?y) ?res) (+ ?x?y))
```

In this case, the result returned by the form `(+ ?x ?y)` will be bound to the variable ?res following execution of the COLANDER form `+(?x, ?y, ?res)`.

The simple LISP primitives expect that all of their input variables will be bound to values before the primitive is invoked. If the result variable of a primitive lisp procedure is bound on entry, then the primitive will succeed only if the calculated result is equal to the given result value.

5.14 Combining Description Files

COLANDER supports partitioning a description into multiple files.

`INCLUDE` filename
`LOAD` filename

To include a file within a language description, use the `INCLUDE` construct. To load a file containing LISP code within a language description, use the `LOAD` construct. The various sections of a description are generally concatenated; however procedure definitions appearing later in a description supersede those given in an earlier-appearing file.

5.15 Hints for Description Writers

Much of the difficulty in writing a language specification comes from the need to design appropriate data representations. Information can either be collected together into larger structures or maintained in separate facts. For instance, consider the representation of bindings and entities in a description. Suppose that one wants to represent a binding as a name/entity pair with a location. This can be defined in any of three ways:

1. As a single fact `bound(name, entity, at)`.
2. As a pair of facts `bound(name, entity)` and `name-bound-at(name,at)`.
3. As a pair of facts `bound(name, entity)` and `entity-bound-at(entity,at)`.

The choice between the approaches can affect efficiency, although the effects are slight in this example. In general, keeping facts small may speed up unification and simplify writing the description, but grouping frequently-referenced information together can speed up access and assertion.

The use of datapools in a language description also requires careful thought. When the correspondence between datapools in the description, and the contours of the language is kept simple, the description seems quite straightforward. However, there are times when contextual information derived from the tree is essential to the description, but such information cannot be placed in the current "scope". For instance, when a tagged variant record is being declared, the type of the tag field is needed within each variant for checking, but this is really not part of a scope. Rather, it is information that must be passed by allocating and propagating a new context datapool.

Chapter 6

Consistency Maintenance

An incremental evaluator for a logical constraint grammar(LCG) associates the goals of an LCG description with subtrees in the syntax tree that represents a document, and then attempts to satisfy the goals associated with each subtree in the syntax tree. When evaluating a logical constraint grammar, changes to the document are ultimately reflected in changes to the contents of the logic database. In response to a change in the document, previously unsatisfied goals may become satisfiable and previously satisfied goals may become unsatisfiable.

Consistency maintenance[1] is the process of keeping the state of the document consistent with the contents of the database. A *consistency manager* performs consistency maintenance. A consistency manager provides the basis for implementing an incremental LCG evaluator. This chapter describes the consistency maintenance as it applies to incremental LCG evaluation.

To maintain consistency between the contents of the database and the document being edited, changes to the database must be carefully monitored. Either the addition of a value to the database or the deletion of a value from the database may require a goal to be retried.

Figure 6.1 illustrates the problem of handling deletions from the database. On the left-hand side of the figure, the assignment of ⟦ 12.0 ⟧ to the variable ⟦ X ⟧ is valid since ⟦ X ⟧ is declared to be **real** within ⟦ Q ⟧. After the declaration of ⟦ X ⟧ in procedure ⟦ Q ⟧ is deleted, as on the right-hand side of the figure, the assignment of the value ⟦ 12.0 ⟧ may or may not be legal, depending on the type declared for ⟦ X ⟧ in procedure ⟦ P ⟧.

Figure 4.5 (page 82) illustrates one possible database for the fragment on the left-hand side of Figure 6.1. When the declaration of ⟦ X ⟧ in ⟦ Q ⟧ is deleted, the consistency manager must remove from "scope datapool 2" any facts representing that declaration. Moreover, any goals that used those facts should be marked undetermined and attempted again. Section 6.3 discusses what must happen when collections or values are removed from the database.

Handling additions to the database is more difficult. Consider the fragment of a Pascal program on the left-hand side of Figure 6.2. This fragment is well-typed, with the

[1] Perhaps "inconsistency maintenance" is a better term, since the document is probably inconsistent most of the time.

```
procedure P;                          procedure P;
   X,Y : ??;                             X, Y : ??;
   procedure Q;                          procedure Q;
      X : real;                             ...
      begin                      ⇒          begin
            X := 12.0;                            X := 12.0;
            ...                                   ...
      end                                   end
```

Figure 6.1: Deletion of Declaration Requires Rechecking

```
procedure P;                          procedure P;
   X, Y : real;                          X, Y : real;
   procedure Q;                          procedure Q;
      begin                                 X : ??;
            ...                             begin
            X := 12.0;             ⇒              X := 12.0;
            ...                                   ...
      end                                   end
```

Figure 6.2: New Declaration Captures Existing Binding

name "X" within procedure ⟦ Q ⟧ resolved to the variable ⟦ X ⟧ declared in procedure ⟦ P ⟧.

If a new declaration for a variable named "X" is inserted, as on the right of Figure 6.2, then the constraints associated with the assignment statement must be rechecked. If the type of the new variable is **real**, then the fragment remains well-typed. However, if the new variable ⟦ X ⟧ was declared to be of type **integer**, a type-inconsistency must be signaled.

Figure 4.5 (page 82). illustrates one possible database for the fragment on the right-hand side of Figure 6.2. Figure 4.5 can be interpreted as representing the contents of the database after the change has occurred. Facts representing the declaration of ⟦ X ⟧ are present in both scope datapools. Prior to the addition of the declaration for ⟦ X ⟧ in ⟦ Q ⟧, the use of "X" in ⟦ Q ⟧ was resolved to the declaration of ⟦ X ⟧ in ⟦ P ⟧. Presumably, the "parent" fact was used to find the declaration of ⟦ X ⟧ in ⟦ P ⟧. Note that only two datapools that represent scopes are involved here. Other scope or context datapools that were not searched when looking for the declaration of ⟦ X ⟧ will not be affected by the addition of the new declaration. Adding the declaration for ⟦ X ⟧ in ⟦ Q ⟧ should trigger the resatisfaction of the goals associated with the assignment statement.

Suppose instead that no declaration for X was visible in Q (prior to adding the new declaration in Q). Then the goals expressing contextual constraints on the assignment statement would not be satisfied. When the new declaration is added to the database, it will enable those goals to become satisfied. Since an incremental evaluator must attempt to satisfy all goals that can be satisfied, all of the goals that can be satisfied will eventually be attempted.

Section 6.1 reviews related work on consistency maintenance. Section 6.2 then

defines the notion of consistency used here. The solution to the problem of handling deletions from the database is discussed in Section 6.3.

Intelligently selecting which goals should be retried after a value is added to the database is crucial to efficient consistency maintenance. The consistency manager must avoid unnecessary computation and unnecessary storage overhead while ensuring that no goals remain satisfied incorrectly. Two different ways to handle additions to the database are described in this chapter. *Holes* (Section 6.4) provide a means for representing data values whose absence from the database was used in satisfying a goal. When a hole is filled, the consistency manager must retry all of the goals that depended upon the absence of that value. *Shadowing rules* (Section 6.5) are inference rules, computed from a static analysis of a LCG description, that help to determine which goals must be attempted again when a value is added to the database. Shadowing rules require less storage but more computation than holes. In Section 6.6, the special cases that arise in a COLANDER description are discussed. Finally, sections 6.7 and 6.8 contain an overview of the implementation of consistency maintenance used in COLANDER.

6.1 Background

Consistency maintenance has been studied in the AI community where it is known as "reason maintenance" or even "truth maintenance" [30, 40, 87]. Reason maintenance systems are concerned with maintaining the validity of inferences within logic-based systems. As premisses are added or removed from the set of premisses considered true, a reason maintenance system updates old inferences and makes new ones. If a change to the set of premisses leads to a contradiction, then the system must detect that a contradiction exists. The system may effect a revision that will eliminate the contradiction as well. Since the reasons for considering an inference valid are considered as important as the actual inferences, reason maintenance systems maintain mappings that allow the system to show the justifications underlying any conclusion.

Reason maintenance systems in many AI applications are *proactive*; when an inconsistency is detected, it is the task of the reason maintenance component to find a way to achieve a consistent state. Most reason maintenance systems use some form of dependency-directed backtracking to restore consistency.

Consistency management for LCGs borrows from reason maintenance systems, but is directed toward the specific application of maintaining the consistency of documents during editing. It differs from reason maintenance systems in four ways.

1. The presence of a structured database complicates the task of detecting when an addition to the database conflicts with an existing value or inference. Adding or deleting a fact in one datapool may require goals that were originally satisfied by matching facts from another datapool to be attempted again.

2. The *closed world assumption* [110] clearly holds for the applications of interest. It is valid to assume that the database contains all of the useful facts that can be known about a document, and that the absence of a datum indicates that the datum does not hold.

3. A complete language description that can be analyzed statically is assumed. The language description provides a closed context in which the forms of all possible goals and data values are known.

4. In a development environment, a document is usually incomplete and rarely has all of its associated goals satisfied. To provide maximum flexibility in editing, the consistency manager should signal inconsistencies to the user but not attempt to remedy those inconsistencies. It is left to the user to add or remove information in order to remove inconsistencies[2].

6.2 Consistency

The task of an incremental LCG evaluator is to ensure that all the goals associated with a document that can be satisfied are satisfied, and to ensure that no unsatisfiable goal is considered satisfied. During editing, every goal associated with a document is in one of three states: *satisfied*, *unsatisfied*, or *undetermined*.

Definition 18 A goal is **satisfied** if it succeeded when it was attempted. A goal is **unsatisfied** if it failed when it was attempted. A goal is **undetermined** if it has not been attempted, or if it has been satisfied but must be reattempted due to changes in the document. Every satisfied goal $\mathcal{G}$ is supported by a possibly empty collection of data values $Supports(\mathcal{G})$. ∎

The elements of $Supports(\mathcal{G})$ are the data values that justify considering $\mathcal{G}$ "satisfied".

Definition 19 A document is **consistent** with the database if and only if

1. For every unsatisfied goal $\mathcal{G}$, attempting $\mathcal{G}$ does not satisfy $\mathcal{G}$, and
2. For every satisfied goal $\mathcal{G}$ with supporting values $Supports(\mathcal{G})$, retrying $\mathcal{G}$ satisfies $\mathcal{G}$ and does not change $Supports(\mathcal{G})$.

A document is **inconsistent** with the database if it is not consistent. ∎

This definition of consistency deals with consistency between the state of the consistency manager (representing the state of the document) and the contents of the database. Definition 19 does not say anything about the consistency of the document *vis á vis* its underlying static semantics. However, in an LCG, it is assumed that all goals should ultimately be satisfied. When a document has unsatisfied or undetermined goals, it is up to the consistency manager to attempt those goals whenever they may become satisfied.

Definition 20 A document is **completely satisfied** if and only if that document is consistent with the database and all of the document's associated goals are satisfied. ∎

A document is assumed to be consistent with respect to its static semantics when it is completely satisfied.

[2]The user can be relieved of part of the task of satisfying the unsatisfied goals by adding an inference component to the consistency manager. This improvement is left for future research.

6.3 Maintaining Consistency in the Presence of Deletions

Values are removed from the database under two conditions: either the subtree that owns the value is removed from the syntax tree, or the that asserted the value is retried during incremental evaluation. When a previously-satisfied goal $\mathcal{G}$ is retried, it is necessary to remove (at least temporarily) all of the values that were created when $\mathcal{G}$ was satisfied originally. Otherwise, the state of the database would be incorrect.

When a value is removed from the database, the consistency manager must retry all those goals that actually depend on that value. If the consistency manager knows the goals that depend on a given value then it is easy to compute the goals that must resatisfied when a value is removed from the database.

The connection between subtrees in the syntax tree and values and collections in the database is established using the notion of *ownership*. An evaluator for a LCG associates instances of goals from the LCG with subtrees in the syntax tree that represents a document. Goals that are attempted during editing are associated with a particular subtree in the syntax tree. Collections or values created when a goal associated with a subtree succeeds are owned by that subtree. Most values and collections in the database are owned by some subtree in the abstract syntax tree. Usually, a value or collection $\mathcal{D}$ is owned by the subtree associated with the goal that created $\mathcal{D}$. However, values and collections created by goals independent of any syntactic structure may not have an owner.

For instance, if the goal $\mathcal{G}$ associated with a subtree N (denoted "N.$\mathcal{G}$") adds a value $\mathcal{D}$ to the database, then $\mathcal{D}$ will be owned by the goal N.$\mathcal{G}$. When the subtree N is deleted, all the data values and collections owned by N must be removed from the database. If $\mathcal{D}$ is a datapool, then $\mathcal{D}$ as well as all of the facts that $\mathcal{D}$ contains must be removed. If $\mathcal{D}$ is an entity, the all of its properties must be removed. By associating values and collections with the instances of the goals that created them, it is straightforward to find the values and collections that must be removed when their owners are removed.

To handle deletions from the database it suffices to maintain the set $Supports(\mathcal{G})$ for each satisfied goal. Recall that for any goal $\mathcal{G}$, $Supports(\mathcal{G})$ is the set of values present in the database that were used in satisfying $\mathcal{G}$. When a value $\mathcal{D} \in Supports(\mathcal{G})$ is removed from the database, the goal $\mathcal{G}$ must be retried.

The elements of $Supports(\mathcal{G})$ can be detected automatically by the interpreter for the logic language. The interpreter has to maintain a stack of the data values that it uses during the interpretation of a goal. The stack is required in order to handle backtracking correctly—one wants to keep only the values actually used to satisfy the goal.

Since the consistency manager ordinarily maps from values and collections to the goals that depend on those values or collections, the inverse mapping can be kept instead.

Definition 21 The set $Justifies(\mathcal{D})$ is the set of all satisfied goals $\mathcal{G}$ that are supported by $\mathcal{D}$. That is, $Justifies(\mathcal{D}) = \{ \mathcal{G} \mid \mathcal{D} \in Supports(\mathcal{G}) \}$. ∎

The set *Justifies* provides an efficient method for finding the goals affected by a deletion once the deletion is detected. The same stack that is used for calculating $Supports(\mathcal{G})$ can be used to calculate $Justifies(\mathcal{D})$. *Justifies* can also be used in handling additions to the database, a topic addressed in subsequent sections.

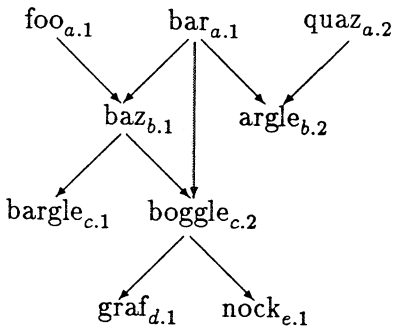


Figure 6.3: Database Dependency Graph

When retrying a goal, the values that were created by that goal (and the values that depended on those values ...) must be removed from the database. Yet since an identical value may be created when the goal is attempted, it is premature at this stage to actually delete those values from the database.

Temporarily retracting values is one technique for hiding values from the interpreter without actually removing those values. When a data value is temporarily retracted, it will not be found in the database by the normal access procedures. However, it is simple to restore the data value if an identical value is created when the goal is resatisfied. In this case the temporarily retracted value is restored, and goals that used the original value remain supported. The process of temporarily retracting values is isomorphic to garbage collection. When a value $\mathcal{D}$ is temporarily removed from the database, then any values created by goals that are supported by $\mathcal{D}$ must be temporarily removed as well. The process of temporarily removing values must proceed recursively until all of the values that were created based on the presence of $\mathcal{D}$ have been visited. Collections do not have to be temporarily removed, because every access to a collection is via a value. Thus it suffices to temporarily remove values only.

Figure 6.3 shows a hypothetical state of the database. Fact values like "foo" and "bar" are subscripted by a N.$\mathcal{G}$ pair indicating the tree node N and goal $\mathcal{G}$ that own the value. For instance, both "foo" and "bar" are owned by goal 1 of node a. The arrows in Figure 6.3 indicate the *Justifies* set for each data value. For instance, "bar" was used during the satisfaction of $b.1$, $b.2$, and $c.2$.

Each data value can be tagged with an integer value indicating the number of times it has been temporarily retracted. If the count field is 0, then the value is not retracted; a non-zero count field indicates that the value should not be found when a literal is matched against the database. Each time a data value can be reached from a temporarily retracted data value by following links in the *Justifies* sets, its retraction count is incremented. For instance, if all of the values owned by $a.1$ are temporarily retracted, the counts associated with each value in Figure 6.3 would be as in Figure 6.4.

Data values are temporarily retracted just before a goal is retried. If the goal being attempted asserts a value that is identical to one previously asserted (now temporarily-

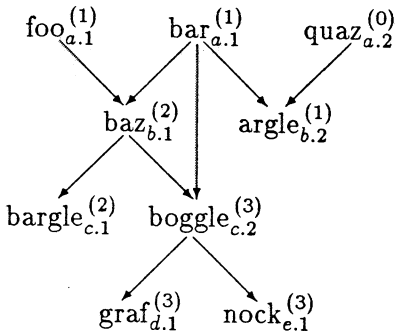


Figure 6.4: Database Dependency Graph with Temporary Retraction Counts

retracted), the temporary retraction is reversed. Suppose $\mathcal{D}$ is temporarily retracted. Then it has a retraction count of n where $n > 0$. The data value $\mathcal{D}$ can be un-retracted by decrementing its count field, along with all of the count fields of values directly or indirectly supported by the value being re-asserted, by n. This restores the count in $\mathcal{D}$ to 0, indicating that $\mathcal{D}$ is no longer temporarily retracted. This algorithm does not re-assert all of the values in the database that were temporarily retracted, however. For instance, in Figure 6.4, if "foo" is re-asserted but "bar" is not, then the count on "baz" will be set to 1. So "baz" continues to be temporarily retracted.

If a data value that was temporarily retracted is not re-asserted, as is the case with "bar" above, then the value must be permanently removed from the database. The values depending on that value need not be removed from the database, but the goals that directly depend on the value being removed must be added to a worklist to be retried later. Continuing the example begun above, if "foo" is re-asserted but "bar" is not, then the goals directly depending on "bar" must be added to the worklist. Those goals are $b.1$, $b.2$, and $c.2$.

The strategy presented here delays adding goals to the worklist as long as possible. For instance, suppose that $b.1$ is re-attempted and "baz" is re-asserted. Then "bargle" would be restored to the database, and $c.1$ would not have to be re-attempted. The consistency manager for *Pan* uses the method of retraction counts to implement temporary retraction of values.

6.4 Holes

To maintain the consistency of the database, one must handle both deletions and additions of information. As discussed in Section 6.3, it is straightforward to handle deletions of information—the system simply recomputes any goals that actually used the information being deleted.

Maintaining consistency in the presence of additions is more difficult. A new data value may allow unsatisfied goals to become satisfied, but it may also invalidate goals previously satisfied. Creating new collections—datapools, entities, or subtrees—does not force any goals to be attempted again. It is only when values become associated with those

collections or refer to those collections that the consistency manager must become involved.

One way to determine which goals to revise is to represent explicitly the values that are found to be "absent" from the database[3]. Representing the absent values requires a way to determine which values are absent along with a method for representing absent values.

When an attempt to find a value $\mathcal{D}$ fails, information about the absence of $\mathcal{D}$ can be retained. All of the goals that are satisfied in spite of attempts to match against $\mathcal{D}$ can be made to depend explicitly on the marker that indicates the absence of $\mathcal{D}$. If the $\mathcal{D}$ is added later, the consistency manager can locate all of the satisfied goals that depend on the absence of $\mathcal{D}$ by using the dependency information associated with the marker. Call the markers representing absent data values **holes**[4].

Since it is unlikely that an absent data value $\mathcal{D}$ will be specified fully, a pattern based on the known and unknown values associated with $\mathcal{D}$ must be used to represent the hole. A pattern represents one or more absent values, depending upon the number of variables in the pattern and the number of possible values that can fill it.

For example, suppose that the binding between a name and an entity is represented by a COLANDER fact of the form "`bound(`*?name*, *?entity*`)`", and suppose that an attempt to match "`bound("X",` *?entity*`)`" in the datapool $\mathcal{P}$ fails. Then a hole representing `bound("X",` *?entity*`)` must be installed in the datapool $\mathcal{P}$ unless such a hole already exists. The pattern used to represent the hole necessarily contains a variable in the second position (*?entity*). This pattern matches an arbitrary number of facts, even if at most one binding between `"X"` and an entity will ever be added to the datapool $\mathcal{P}$.

Representing absent values works, provided that the holes are detected and represented correctly. While potentially expensive in storage, properly implemented it effectively characterizes the goals that must be reconsidered after the addition of any new value. Since holes are represented using patterns, using holes is a conservative strategy. Holes can represent multiple data values, so filling a hole with a particular value may cause goals that do not require that particular value to be retried.

The storage cost of holes is potentially high, particularly for languages having nested block structure. If the visibility rules of the target language require searching the surrounding scopes for a binding, and if the binding is found at the outermost level of nesting, then a hole for the fact that represents the binding would be created in each of the surrounding scopes. Since uplevel references are often resolved to the outermost scope, the proliferation of holes is a real possibility. At worst, every datapool that represents a scope would contain a hole for each global variable in the program.

Another simple approach—and one that is also clearly correct—is to recheck all goals after a new value is added to the database. This approach minimizes storage overhead at the cost of recomputation.

The above two approaches—representing all of the absent data and retrying all of the goals at each change—define a continuum over which storage and processing can be traded. Over this continuum, the information present in a language description can be used

[3]This is similar to the technique (used in reason maintenance systems) of representing values that are assumed to be false.

[4]This differs from the normal definition of a "hole in a scope".

to derive rules that determine which goals may need to be revised in the presence of added information.

The next section introduces **shadowing rules**, a technique for inferring which goals must be attempted again when a value is added to the database.

6.5 Shadowing and Shadowing Rules

Shadowing rules are inference rules associated with data literals that help to determine which values may be shadowed[5] when another value is added to the database. The goals that depend on the shadowed values are then retried. Shadowing rules are created off line, by examining the complete language description when the language description is compiled.

For example, if the binding between a name and an entity is represented by a fact of the form "bound(*?name*, *?entity*)", then a simple shadowing rule would indicate that a new binding for a specific name shadows the other bindings of that name:

$$< \texttt{bound}(?name, ?entity1), ?scope1 > :: < \texttt{bound}(?name, ?entity2), ?scope2 >$$

The appearance of *?name* in both the source (left-hand side) and the target (right-hand side) of the rule constrains the first argument of the two facts to be identical, while different variable names permit alternate matchings. Thus, "<bound("foo", entity1), dp1>" could shadow "<bound("foo", entity2), dp2>" or even "<bound("foo", entity1), dp1>", but not "<bound("bar", entity3), dp3>".

Shadowing rules have three components: a source, a target, and a constraining clause. The source and target are literals that represent data values. The constraining clause is a (possibly empty) literal clause that can be used to further constrain the data values selected by the shadowing rule. The notation for shadowing rules is

$$C_{source} :: C_{target} \text{ :- } C_1, \ldots, C_n$$

which can be read as "C_{source} shadows C_{target} when C_1 and ... and C_n all hold." C_{source} is a literal that represents the data value being added to the database, C_{target} represents the data value that is shadowed, while $C_1, \ldots, C_n$ is a logic goal that must be satisfied for the shadowing to occur. If $n = 0$ in the constraining clause, the shadowing rule is abbreviated to $C_{source} :: C_{target}$.

When a shadowing rule is checked, the literal C_{source} is unified with the data value being added to the database to get a substitution θ. If this unification succeeds, then the goal "$\leftarrow (C_1, \ldots, C_n, C_{target})\theta$" is attempted just like a normal goal. If the entire clause succeeds, then C_{target} is shadowed by C_{source}, and the goals that used C_{target} must be retried. Since C_{target} is a data literal, there will be a dependency record tagged onto it that indicates the goals that depend on that value. (This is just the set *Justifies* discussed in Section 6.3.) All solutions to the goal "$\leftarrow (C_1, \ldots, C_n, C_{target})\theta$" must be considered[6].

[5]The use of the term "shadow" here conflicts slightly with the definition used by Garrison [47]. In his work, shadowing occurs between bindings in the language being described. Here, shadowing occurs between elements of a description of the target language.

[6]Shadowing rules can be implemented as a special kind of database trigger as described in Section 5.9.

To understand shadowing rules, it is necessary to understand how they are used. When a new value $\mathcal{D}$ is added to a datapool, the shadowing rules for $\mathcal{D}$ determine which goals might have to be retried. Goals are found indirectly, by locating data values shadowed by $\mathcal{D}$. If the new value $\mathcal{D}$ shadows the data value $\mathcal{D}'$, then the goals in *Justifies*($\mathcal{D}'$) may have to be retried. The problem of shadowing is thus reduced to shadowing between data values. Additionally, the system is able to to use the *Justifies* mapping from data values to the goals that the values support.

Shadowing rules act as filters on the choice of which goals to revise. The coarsest possible filter corresponds to re-attempting all of the goals whenever a new data value is asserted. The finest possible filter corresponds to storing the holes as well as the values actually asserted by the goals. Shadowing rules can provide filters with intermediate granularities. When a new data value is added to the database, the shadowing rules for that value select a series of other data values that might be shadowed by the new addition. Naturally, the finest filter that is consistent with low storage and computational requirements and that ensures that any goal possibly affected by the new value will be retried should be used.

Consider again the shadowing rule given by

$$< \texttt{bound}(?name, ?entity1), ?scope1 > :: < \texttt{bound}(?name, ?entity2), ?scope2 >$$

A similar rule having a coarser mesh is

$$< \texttt{bound}(?name1, ?entity1), ?scope1 > :: < \texttt{bound}(?name2, ?entity2), ?scope2 >$$

Here, the distinction between *?name1* and *?name2* indicates that the addition of *any* new fact named bound will shadow *all* of the facts named bound everywhere in the database.

Even though a value $\mathcal{D}'$ is shadowed, the goals in *Justifies*($\mathcal{D}'$) may not have to be retried. For instance, suppose that the rule

$$< \texttt{bound}(?name, ?entity1), ?scope1 > :: < \texttt{bound}(?name, ?entity2), ?scope2 >$$

is being used in conjunction with the example of Figure 6.2 (page 110) . Suppose, too, that the fact "bound("X", entity12)" is being added to the datapool that represents the scope for procedure Q. (This is "scope datapool 2" in Figure 4.5 on page 82.) When the new fact is added to the database, all facts of the form "bound("X",*?entity*)" are selected by the shadowing rule. However, any goals that did not examine "scope datapool 2" need not be retried.

More generally, suppose a new data value $\mathcal{D}$ being added to a datapool $\mathcal{P}$ shadows an existing value $\mathcal{D}'$ on which the goal $\mathcal{G}$ depends. Then $\mathcal{G}$ must be attempted again only if the datapool $\mathcal{P}$ was searched when $\mathcal{G}$ was satisfied the first time.

Using shadowing rules efficiently requires that the datapools actually searched when attempting a goal be retained on a per-goal basis. This observation accounts for the extra information retained by the consistency manager for handling additions to the database: *for each satisfied goal, the consistency manager retains the set of datapools searched while satisfying that goal.*

The sequence of datapools accessed when satisfying a goal can be represented as a path through the database where each element on the path is a datapool. Moreover, because

of the structure of programming languages, most goals in a single scope have similar paths—almost always a prefix of a single general path. With an appropriate data representation, overlapping the paths leads to considerable sharing.

One possible data structure for storing paths is a an adaptation of Tarjan's linking/cutting trees [129]. The sequence of datapools in a path is represented using a tree. Within a tree, a path leads from a node (internal or external) in the tree towards the root. A single path can be represented using a pair of pointers to the first and last nodes in the sequence. This representation is used in COLANDER.

6.5.1 Generating Solutions to Goals

Shadowing rules are created at language-definition time by analyzing the language description. Each goal associated with a syntactic structure in the language creates potential shadowing. To calculate shadowing rules, each goal clause is expanded into a finite ordered collection of possible *solution schemata*. Solution schemata are generated in the same order that they would appear during evaluation of the clause. Shadowing rules are calculated from a pairwise analysis between values appearing in different solution schemata generated from the same initial goal. Shadowing arises when the addition of a value to the database enables a solution schema $\mathcal{S}$ to succeed. In that case, solutions derivable from schemata following $\mathcal{S}$ that may have succeeded must be re-attempted. Calculating shadowing rules is a two phase process: generating sets of solution schemata from goal clauses and analyzing those sets for possible shadowing.

The presentation of logic programming in Section 4.3 hints that the successive goal clauses in a derivation from an initial goal $\mathcal{G}$ can be enumerated. In brief, the technique is to treat the procedures in a logic program like productions in a generative grammar. Using the initial goal as a starting point, one can successively replace procedure heads with procedure bodies. The procedure heads function like nonterminal symbols while data literals function like terminal symbols. A solution schema corresponds to a sentential form. Solution schemata are derived by repeatedly replacing literals that match the heads of program clauses with the bodies of those clauses.

The generation of solution schemata mimics the backtracking search used by a PROLOG interpreter. The order of the various clauses that define a procedure is important; during the derivation of a solution schema, that ordering is preserved. Data literals are never replaced when enumerating solution schemata, nor are literals that denote maintained subtree properties. Rather than expanding maintained properties, each use of a maintained property is treated as an unexpandable literal. Each definition for a maintained property can be analyzed as a separate goal.

Figure 6.5 provides an example of solution schemata being generated from a single goal. The literals `F1`, `F2`, `F3`, and `F4` represent data values. There are four procedures as well, two of which have alternative clauses. Starting from the goal, a sequence of solution schemata is enumerated. For instance, the first solution schema is derived from the goal by first replacing the goal literal `P`(*?A*, *?B*) with the first clause in the definition of `P`, and then replacing the goal literal `R`(*?C*) with the first clause in the definition of `R`. The example is contrived in order to illustrate various points that will be presented in the rest of this chapter.

Facts:

F1(?X, ?Y) F2(?X, ?Y, ?Z)
F3(?X) F4(?X, ?Y)

User-Defined Procedures:

P(?X, ?Y) :- S(?X), =(?X, ?Z), F1(?Z, ?Y).
P(?X, ?Y) :- F2(?X, ?Y, ?Z), F3(?Z), P(?X, ?Z).
R(?Y) :- F3(?Y).
R(?Y) :- F4(?Y, constant).

Primitive Procedures:

Q(?X, ?Y).
S(?X).

Goal:

:- P(?A, ?B), Q(?B, ?C), R(?C).

Solution Schemata:

$\mathcal{S}_1$ = S(?A), =(?A, ?v1), F1(?v1, ?B), Q(?B, ?C), F3(?C)
$\mathcal{S}_2$ = S(?A), =(?A, ?v1), F1(?v1, ?B), Q(?B, ?C), F4(?C, constant)
$\mathcal{S}_3$ = F2(?A, ?B, ?v2), F3(?v2), S(?A), =(?A, ?v3),
F1(?v3, ?v2), Q(?B, ?C), F3(?C)
$\mathcal{S}_4$ = F2(?A, ?B, ?v2), F3(?v2), S(?A), =(?A, ?v3),
F1(?v3, ?v2), Q(?B, ?C), F4(?C, constant)
$\mathcal{S}_5$ = F2(?A, ?B, ?v2), F3(?v2), F2(?A, ?v2, ?v4), F3(?v4),
P(?A, ?v4), Q(?B, ?C), R(?C)
$\mathcal{S}_6$ = ...

Figure 6.5: Solution Schema Generated from a Goal

The derivation of a solution schema can be viewed as the top-down construction of a derivation tree analogous to the derivation trees of a context-free grammar. The frontier of the tree describes the solution schema derived. At any given point in the derivation of a solution schema, one particular procedure literal is *eligible* for expansion and there is a single program clause that can be used to expand it. In PROLOG, the eligible literal is the leftmost literal.

Each internal node in the derivation tree of a solution schema represents a program clause that was used to derive the children of that node. The "root node" is associated with the unique clause that denotes the initial goal. The path from the root to the eligible literal describes a sequence of program clauses $[\rho_1, \ldots, \rho_k]$ that were applied to reach this stage in the derivation. Call this path an *expansion sequence*.

Definition 22 An **expansion sequence** is the sequence of program clauses described by the path in the derivation tree of a solution schema from the root to a literal in the tree. ∎

During solution schema generation the choice of the next literal to expand is modified as follows to avoid nonterminating expansions.

Definition 23 An expansion sequence $[\rho_1, \ldots, \rho_k]$ **N-loops** if and only if there are N adjacent identical subsequences in the sequence. If replacing a literal by a program clause causes the expansion sequence determined by the path from the root to that literal to N-loop then the entire solution schema N-loops. ∎

For example, the expansion sequence [a, b, c, b, c, b, c] 3-loops.

Definition 24 An enumeration of solution schemata is **N-complete** if it includes all the solution schemata that N-loop, together with those solution schemata preceding the solution schemata that N-loop. ∎

If a solution schema in an N-complete enumeration does not N-loop, then it does not contain any literals that can be expanded by procedure clauses. For example, in Figure 6.5, solution schemata $\mathcal{S}_1$–$\mathcal{S}_5$ comprise the 2-complete enumeration for the given goal. $\mathcal{S}_1$ does not 2-loop. It is composed of primitive functions and data literals. The solution schema $\mathcal{S}_6$ is computed using a 3^{rd} expansion of procedure P. Using N-complete enumerations limits the number of solution schemata derived from a single goal. In calculating shadowing rules, 2-complete enumerations are used.

The formal definition of the leftmost eligible literal can now be presented.

Definition 25 The **leftmost eligible literal** in a schema is the leftmost literal term C meeting the following two criteria:

1. C does not represent a fact or a property,
2. Expanding C by the chosen procedure clause does not cause the schema to N-loop, where N is a parameter of the generation scheme.

∎

Substituting a literal with a procedure body follows the normal strategy for expanding a literal during execution of a logic program. The literal terms in a solution schema include literals representing data, system-defined procedures, or unexpanded user-defined procedures.

Definition 26 If the current goal is "$\leftarrow C_1, \ldots, C_k$", a **schema derivation step** consists of selecting the leftmost eligible C_j, unifying it with the head of some procedure clause "$A \leftarrow B_1, \ldots, B_n$" to obtain a substitution θ, and generating a new goal

$$\leftarrow (C_1, \ldots, C_{j-1}, B_1, \ldots, B_n, C_{j+1}, \ldots, C_k)\theta$$

∎

Substituting a procedure clause that is a unit clause creates a new goal containing fewer literal terms.

Definition 27 A **solution schema** for a goal $\mathcal{G}$ is a goal clause "$\leftarrow C_1, \ldots, C_m$" derived from $\mathcal{G}$ by zero or more schema derivation steps. A solution schema "$\leftarrow C_1, \ldots, C_m$" is denoted by $\mathcal{S} = C_1, \ldots, C_m$. To indicate that $\mathcal{S}$ is a solution schema derived from $\mathcal{G}$, we shall write $\mathcal{G} \models \mathcal{S}$. ∎

The solution schemata derived from a single initial goal are ordered by the derivation procedure. Literals are selected from left to right in a goal; for each literal, the procedure clauses used to replace that literal are ordered within the procedure definition and are used in their defined order.

Definition 28 If $\mathcal{G} \models \mathcal{S}_1$ and $\mathcal{G} \models \mathcal{S}_2$ are two solution schemata derived from the same goal $\mathcal{G}$, then $\mathcal{S}_1$ **precedes** $\mathcal{S}_2$ (denoted $\mathcal{S}_1 \prec \mathcal{S}_2$) if and only if $\mathcal{S}_1$ was generated prior to $\mathcal{S}_2$ during the enumeration of solution schemata from $\mathcal{G}$. ∎

Handling of Variables within Solution Schemata

The more logical variables that are shared between the source, target, and constraining clause literals of a shadowing rule, the fewer the data values that will be shadowed. Variables that are shared between the source and the target of a shadowing rule lead to better computational performance. This fact should be clear from examining the two shadowing rules

$$< \text{bound}(?name, ?entity1), ?scope1 > :: < \text{bound}(?name, ?entity2), ?scope2 >$$

and

$$< \text{bound}(?name1, ?entity1), ?scope1 > :: < \text{bound}(?name2, ?entity2), ?scope2 >$$

In the first rule, the variable *?name* is shared between the source and the target. The first rule selects fewer data values than the second.

The source and the target literals in a shadowing rule are drawn from different solution schema. The logical variables appearing in the original goal are called the **original goal variables** of the goal. The variables that might be common to the two literals are the original goal variables of the original goal.

A solution schema is an expanded version of an initial goal. When the goal is attempted, all of the variables in the goal are initially unbound; as the literals in the goal are attempted, variables become bound to other expressions.

As solution schemata are derived from the initial goal, a series of substitutions is created. Each substitution binds the unbound variables in the eligible literal to expressions in the head of the program clause being used for the expansion.

New variables introduced by expanding a procedure literal are made unique to the entire set of solution schemata for a particular goal. This ensures that if two solution schemata share a variable, then either that variable appeared in the original goal or it was introduced by the same clause of a procedure definition.

When creating the shadowing rules, it is necessary for the derived solution schemata to be expressed in terms of the original goal variables of the initial goal along with the new variables introduced in the derivation process. When a original goal variable is bound to another variable during expansion, either variable (name) can be used in result. In this case, the name of the original goal variable is used.

When a original goal variable is bound to a non-variable expression appearing in the head of a program clause, matters become more interesting. Simply substituting the

expression for the variable would eliminate the original input variable from the solution schema and would hide the fact that the binding between a variable and a value took place. To ensure that the binding between the value and the original goal variable is explicitly represented in the solution schema, a simple semantics-preserving program transformation can be applied. Suppose that a variable appearing in the eligible literal would be bound to a non-variable expression E when the literal is unified with the head of a program clause. Then the program clause must be of the form $\mathsf{P}(\ldots,E,\ldots) \text{ :- } C_1,\ldots,C_n$. Such a clause can be transformed into the equivalent clause $\mathsf{P}(\ldots,X,\ldots) \text{ :- } X = E, C_1,\ldots,C_n$ where X is a new variable not appearing in the original program clause, and "=" denotes the unification operator. This transformation makes the unification between the original goal variable and the expression explicit. During the analysis of shadowing, this transformation can be applied as necessary.

6.5.2 Definition of Shadowing

Shadowing occurs when a value added to the database might invalidate a goal that was satisfied previously. Shadowing is defined between two data values—values that can be added or removed from a datapool. During execution, a value in the database is matched by a data literal.

When a goal $\mathcal{G}$ is attempted, the solution schemata $\mathcal{S}_1,\ldots,\mathcal{S}_n$ derived from $\mathcal{G}$ are attempted in the order that they were generated. If $\mathcal{G}$ succeeds, one of the solution schemata (or a solution derivable from it) succeeded while all of the preceding solution schemata (and the solutions derivable from them) failed. Suppose $\mathcal{S}_j$ is the solution schema associated with the attempt that succeeded. Shadowing occurs when a solution schema $\mathcal{S}_j$ has been satisfied but a new data value is added to the database that might enable a solution schema $\mathcal{S}_i \prec \mathcal{S}_j$ to be satisfied. The added value matches a literal C_{im} in $\mathcal{S}_i$. Once the data value matching C_{im} has been added to the database, it is possible that retrying $\mathcal{G}$ would result in $\mathcal{S}_i$ rather than $\mathcal{S}_j$ succeeding[7]. In this case, the literal C_{im} **shadows** the literal terms in $\mathcal{S}_j$. When shadowing occurs, the consistency manager must remove (temporarily or permanently) all of the facts that ensued from the satisfaction of $\mathcal{S}_j$ and then attempt to resatisfy $\mathcal{G}$.

The above definition is much too general for efficient implementation. In the next section, a formal definition for shadowing is presented that reduces the number of potentially shadowed values.

6.5.3 Definition of Shadowing Rules

Shadowing rules are constructed by a pairwise analysis of the solution schemata arising from a single goal. Suppose that a new data value matching C_{ik} is being added to the database. The presence of C_{ik} might allow solution schema $\mathcal{S}_i$ (or a solution derivable from it) to succeed where it failed earlier. Thus, adding the value C_{ik} should force goals that have already succeeded to be attempted again. Goals that have have failed or are undetermined will be attempted in the normal course of incremental evaluation.

This section presents the basic definition of shadowing rules. The rules computed using this definition can be improved in a number of ways in order to limit the number of

[7] If $\mathcal{G}$ failed, then all it solution schema also failed, and the evaluator must retry $\mathcal{G}$ at a later time.

values actually shadowed by an addition to the database. Section 6.5.7 discusses ways to improve the set of shadowing rules computed for a language description.

Handling Variables within Shadowing Rules

When calculating shadowing rules, only those literals appearing in the solution schemata following S_i which have variables in common with C_{ik} are considered. When the same variables appear in both the source and target, a more refined set of shadowing rules is created.

When a solution schema is generated, the set of variables that appear in each literal can be easily calculated. The set of variables bound by each literal can be calculated as well. The definition of shadowing uses the following definitions concerning the appearance and use of variables in a literal.

Calculating which variables are bound by a literal requires that the input/output behavior of the literal be known. Every literal in a language description has an input/output behavior. The input/output behavior describes which variables become bound if the literal succeeds, assuming that other variables were bound before the literal was attempted.

Definition 29 A literal is **determinate** if all of the variables appearing in the literal are guaranteed to be bound to ground terms following successful evaluation of the literal. A literal that is not determinate is said to be **indeterminate.** ∎

In PROLOG and in COLANDER all data literals are determinate. In addition, all of the system procedures in COLANDER are determinate. COLANDER requires that the author of a language description explicitly supply the input/output behavior for any indeterminate procedure appearing in the description. This restriction is not severe in practice; indeterminate procedures do not arise often.

Knowing the input/output behavior of a literal makes it possible to calculate which literal in a solution schema binds a given variable. The input/output behavior of procedures is of general interest in compiling logic programs [31, 92].

When analyzing a goal or a solution schema, it can be ascertained whether the variables appearing in a literal term will act as inputs or outputs to that literal.

Definition 30 The variables appearing in a literal within a given goal can be classified as *input*, *output*, or *undetermined*.

- An **input variable** in a literal is a variable that is bound to a value before the literal is expanded or matched during interpretation.
- An **output variable** in a literal is a variable that is not an input variable, but is bound to a value after the literal is expanded or matched during interpretation.
- An **undetermined variable** in a literal is a variable that is not known to be either an input or an output variable.

∎

Definition 31 Given a solution schema $S_i = C_{i1}, \ldots, C_{im}$, the notation $appears(C_{ij})$ denotes the set of variables that appear in literal C_{ij}, $bound(C_{ij})$ denotes the set of variables

that are bound during the evaluation of the literal C_{ij}, and $\mathcal{V}(\mathcal{S}_i)$ denotes the set of variables appearing in $\mathcal{S}_i$. Formally, for any literal C_{ij} in a solution schema $\mathcal{S}_i = C_{i1}, \ldots, C_{im}$ $(j \leq m)$ let

$$\begin{array}{lcl} \mathit{appears}(C_{ij}) & = & \{\, v \mid v \text{ is a variable appearing in } C_{ij} \,\} \\ \mathit{bound}(C_{ij}) & = & \{\, v \mid v \text{ is a variable bound when } C_{ij} \text{ succeeds} \,\}, \text{ and} \\ \mathcal{V}(\mathcal{S}_i) & = & \bigcup_j \mathit{appears}(C_{ij}) \end{array}$$

Consider the schema $\mathcal{S}_1 = \mathtt{S}(?A), \mathtt{=}(?A, ?v1), \mathtt{F1}(?v1, ?B), \mathtt{Q}(?B, ?C), \mathtt{F3}(?C)$ of Figure 6.5. Suppose that S is a primitive procedure (not a data literal) that binds $?A$. The primitive procedure $=(?A, ?v1)$ will bind the value of $?A$ to $?v1$. Since all data values are ground terms, matching a data literal with a value causes any unbound variables in the literal to become bound. So it is the case that $\mathit{appears}(\mathtt{F1}(?v1, ?B)) = \{\, ?v1, ?B \,\}$ and $\mathit{bound}(\mathtt{F1}(?v1, ?B)) = \{\, ?B \,\}$. The set $\mathcal{V}(\mathcal{S}_i)$ is given by $\mathcal{V}(\mathcal{S}_i) = \{\, ?A, ?B, ?C, ?v1 \,\}$. The variables in the original goal $\mathcal{G}$ is given by $\mathcal{V}(\mathcal{G}) = \{\, ?A, ?B, ?C, \,\}$.

It is possible that the original goal variables of a goal do not appear in data literals but do participate in shadowing. This arises when a variable is bound in a literal that does not represent a data value (e.g., a procedure literal) and that variable is later used to bind another original goal variable. For example, this occurs with the variable $?A$ in $\mathcal{S}_1$. The variable $?A$ is bound by the primitive procedure S and is then used to bind $?v1$ and ultimately $?B$, but $?A$ does not appear in a literal representing a data value.

The dependency relation $\mathit{fnd}(X, Y)$ (pronounced "X founds Y") formalizes the notion of one variable being used to define the value of another.

Definition 32 The dependency relation $\mathit{fnd}(X, Y)$ holds whenever X is an input variable to the literal that binds Y. The reflexive, transitive closure of fnd is $\mathit{fnd}^*(X, Y)$:

$$\mathit{fnd}^*(X, Y) = \mathit{fnd}(X, Y) \cup \{\, Z \mid \mathit{fnd}(X, Z) \wedge \mathit{fnd}^*(Z, Y) \,\}$$

Only the original goal variables that are shared between two solution schemata are of interest during the analysis for shadowing. The set $\mathit{appears}^*(C)$ is composed of all of the original goal variables of the schema appearing in the literal C, together with all of the those variables bound in C whose values depend on original goal variables that do not appear in C.

Definition 33 Suppose $\mathcal{G} \models \mathcal{S}_i = C_{i1}, \ldots, C_{ik}$. The set of original goal variables of $\mathcal{S}_i$ is given by $\mathcal{V}(\mathcal{G})$. Then

$$\begin{array}{lcl} \mathit{appears}^*(C_{ij}) & = & \{\, X \mid X \in (\mathit{appears}(C_{ij}) \cap \mathcal{V}(\mathcal{G})) \,\} \cup \\ & & \{\, Y \in \mathcal{V}(\mathcal{G}) \mid \exists X \in \mathit{appears}(C_{ij}), X \notin \mathcal{V}(\mathcal{G}) \text{ such that } \mathit{fnd}^*(Y, X) \,\} \end{array}$$

For example, in $\mathcal{S}_1$ of Figure 6.5 $appears^*(\texttt{F1}(?v1, ?B)) = \{ ?A, ?B \}$.

Attention can be further limited to data literals that are the *leftmost* data literal in which a given variable appears. It suffices to consider only the leftmost data literal since, if the goal succeeded in that solution schema, then that value is present in the database and has the correct dependency information. Any data literal to the right of the leftmost one will have the same dependency information, and so would lead to redundant rules. This is such an important restriction that it is embedded in the basic definition for shadowing.

Two more definitions are required before the full definition of shadowing can be given. The prefix of a solution schema $\mathcal{S}$ relative to a set of variables V is the clause composed of all those literals which help to define the variables in V. Definition 34 is used in defining the prefix of a schema. It formalizes the notion of one literal depending on another.

Definition 34 Let C_{ij} and C_{ik} be two literals in a goal clause $\mathcal{S}_i = C_{i1}, \ldots, C_{in}$ where $j \leq k$. Then C_{ik} **bdepends** on C_{ij} if C_{ik} uses a variable that is bound in C_{ij}. This relation is denoted by $C_{ij} \lhd C_{ik}$. Formally, $C_{ij} \lhd C_{ik}$ if and only if $bound(C_{ij}) \cap appears(C_{ik}) \neq \emptyset$. Denote the reflexive, transitive closure of $\lhd$ by $\lhd^*$. ∎

Definition 35 The **prefix** of a clause $\mathcal{S} = C_1, \ldots, C_n$ relative to a set of variables V is given by $prefix(S, V) = C_{i_1}, \ldots, C_{i_m} (i_1 < i_2 < \ldots < i_m = k)$ where a literal C_{i_j} appears in $prefix(S, V)$ if and only if

1. $bound(C_{i_j}) \cap V \neq \emptyset$ or
2. There exists a literal C_{i_k} such that $C_{i_j} \lhd^* C_{i_k}$ and $bound(C_{i_k}) \cap V \neq \emptyset$.

∎

Continuing the running example, $prefix(\mathcal{S}_1, \{ ?B \})$ is the clause

$$\texttt{S}(?A), =(?A, ?v1), \texttt{F1}(?v1, ?B)$$

When the same original goal variable is bound identically in the two schemata, then it must be bound to the same value when the two schemata are attempted. When a variable $?X$ appears in two solution schemata $\mathcal{S}_i$ and $\mathcal{S}_j$ generated from the same goal and $prefix(\mathcal{S}_i, \{ ?X \})$ can be unified with $prefix(\mathcal{S}_j, \{ ?X \})$ then $?X$ represents the same value in all solutions that can result from either schema. In this case, $?X$ is **bound identically** in the two solution schemata. Conversely, if $?X$ is a original goal variable that appears in both the $\mathcal{S}_i$ and $\mathcal{S}_j$, but $prefix(\mathcal{S}_i, \{ ?X \})$ cannot be unified with $prefix(\mathcal{S}_j, \{ ?X \})$, then the variable may take on a different values in the two solution schemata.

The formal definition of shadowing can now be presented.

Definition 36 Suppose that $\mathcal{S}_i = C_{i1}, \ldots, C_{im}$ and $\mathcal{S}_j = C_{j1}, \ldots, C_{jn}$ are two solution schemata for a goal $\mathcal{G}$ and $\mathcal{S}_i \prec \mathcal{S}_j$. Then C_{ik} **simply shadows** a literal C'_{jl} derived from a literal C_{jl} in $\mathcal{S}_j$ if and only if

1. Both C_{ik} and C_{jl} denote data values,
2. The set $SS = appears^*(C_{ik}) \cap appears^*(C_{jl}) \neq \emptyset$
3. C_{jl} is the leftmost data literal appearing in $\mathcal{S}_j$ for at least one of the variables appearing in SS, and

F3(*?C*) :: F3(*?C1*)
F3(*?C*) :: F4(*?C2*, constant)
F1(*?v1*, *?B*) :: F2(*?A*, *?B*, *?v2*)
F4(*?C*, constant) :: F3(*?C3*)
F4(*?C*, constant) :: F4(*?C4*, constant)

Figure 6.6: Some Simple Shadowing Rules from the Running Example

4. C'_{jl} is obtained from C_{jl} by uniformly replacing all of the original goal variables of $\mathcal{G}$ appearing in C_{jl} that are not bound identically in $\mathcal{S}_i$ and $\mathcal{S}_j$ with new variables that do not appear in either $\mathcal{S}_i$ or $\mathcal{S}_j$.

∎

The construction of the solution schemata ensures that if $appears^*(C_{ik}) \cap appears^*(C_{jl}) \neq \emptyset$, then each variable contained in that set must appear in $\mathcal{G}$. Figure 6.6 shows some of the simple shadowing rules generated for the goal shown in Figure 6.5.

The definition of shadowing shows why 2-complete enumerations of solution schemata suffice for generating shadowing rules. Shadowing analysis is concerned with interactions between solution schemata. In a 2-complete solution, each case of a procedure definition will be expanded twice, ensuring that each possible interaction is exposed. Expanding the cases a third time cannot possibly add any new interactions. The analysis of shadowing, then, can be restricted to 2-complete enumerations, and can also be restricted so that the actual solution schemata that 2-loop are examined for literals to be shadowed, but need not be examined as sources of shadowing.

It is possible that an original goal variable does not appear in any data literal in a schema. (This occurs with the variable *?A* in $\mathcal{S}_1$.) When this occurs, a potential cause of shadowing may be missed during shadows analysis. The situation can be corrected by modifying the original goal (or some other procedure that was expanded during the generation of the solution schema) to assert a fact that uses those variables that did not appear in any data literal. COLANDER detects this situation and allows the author of the LCG to modify the description as appropriate.

Finally, note that once a variable has been incorporated into a shadowing rule, it is not necessary to examine other uses of that variable within a solution schema. Thus the algorithm for calculating shadows rules needs to process each original goal variable in the shadowed solution schema only once.

6.5.4 Constraining Clauses in Shadowing Rules

The definition of simple shadowing (Definition 36), while correct, can be improved in several ways. There are many tradeoffs in generating shadows rules; some are not yet completely resolved. Obviously, one wants to minimize the absolute number of rules to be checked when a value is added. Beyond that, the actual rules that are created can contain more information than just the source and target literals. Yet complex rules can give rise to extra backtracking, increasing computation time and not providing as fine a filter as

simpler rules. Unbound variables in a shadowing rule force backtracking over all of the possible values for those variables. Variables appearing in the source literal are bound to ground terms when the rule is checked. Any variables that are not bound in either the source or the constraining clause cause backtracking.

Constraining clauses—clauses in the shadowing rule that may reduce the number of goals nominated to be attempted again—are derived using *projections* of the original source solution schema. A projection of a clause selects a subset of the literals appearing in that clause. For improving shadows analysis, two projections are of interest: the simple projection, and the complete projection. The simple projection selects out all of the literals in which the given set of variables appears.

Definition 37 The **simple projection** of a goal clause $\mathcal{S} = C_1, \ldots, C_n$ relative to a set of variables V is defined by $\mathcal{S} \downarrow_V = C_{i_1}, \ldots, C_{i_k} (i_1 < i_2 < \ldots < i_m)$ such that C_{i_j} is a literal of $\mathcal{S} \downarrow_V$ if and only if $appears(C_{i_j}) \cap V \neq \emptyset$. ∎

Consider the solution schema $\mathcal{S}_1$ of Figure 6.5:

$$\mathcal{S}_1 = \texttt{S}(?A), \texttt{=}(?A, ?v1), \texttt{F1}(?v1, ?B), \texttt{Q}(?B, ?C), \texttt{F3}(?C)$$

Then $\mathcal{S}_1 \downarrow_{\{?B\}} = \texttt{F1}(?v1, ?B), \texttt{Q}(?B, ?C)$.

Projections can be regarded as ordered sets of literal terms so that the set difference operator can be applied to them. Thus if $\mathcal{S} \downarrow_V = C_1, \ldots, C_n$ the notation $\mathcal{S} \downarrow_V - \{C_i\}$ denotes the clause $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$.

It must be the case that if C_{ik} actually shadows C_{jl}, then the other literals in $\mathcal{S}_i$ must also succeed. In particular, the literals in $\mathcal{S}_i \downarrow_{bound(C_{jl})}$ must succeed. Constraining the shadowing rule to require that the literals in the simple projection also succeed can lead to improved results by eliminating some false starts that would otherwise be attempted.

Definition 38 Suppose that $\mathcal{S}_i = C_{i1}, \ldots, C_{im}$ and $\mathcal{S}_j = C_{j1}, \ldots, C_{jn}$ are two solution schemata to a goal $\mathcal{G}$, and $\mathcal{S}_i \prec \mathcal{S}_j$. Then $C_{ik} :: C_{jl} \text{ :- } (\mathcal{S}_i \downarrow_{bound(C_{jl})} - \{C_{ik}\})$ if and only if

1. C_{ik} simply shadows C_{jl} and
2. The goal formed by $(\mathcal{S}_i \downarrow_{bound(C_{jl})} - \{C_{ik}\})$ is also satisfied.

The goal clause $(\mathcal{S}_i \downarrow_{bound(C_{jl})} - \{C_{ik}\})$ becomes the **constraining clause** of the rule. ∎

Figure 6.7 shows some of the shadowing rules generated for the goal of Figure 6.5 when the simple projection is used.

The simple projection includes all of the literals in which a variable appears, but it is not necessarily complete. For instance, assuming that variables are bound in left-to-right order, the literal F1(*?v1*, *?B*) binds *?B* but does not bind *?v1*. The complete projection on a solution schema $\mathcal{S}$ relative to a set of variables V includes not only the literals in the simple projection, but also all of those literals on which the variables in V ultimately depend. Using the complete projection may eliminate some undesirable backtracking by assuring that more variables in the constraining clause or target literal are bound.

```
F3(?C) :: F3(?C1)            :- Q(?B, ?C).
F3(?C) :: F4(?C2, constant)  :- Q(?B, ?C).
F1(?v1, ?B) :: F2(?A, ?B, ?v2) :- S(?A), =(?A, ?v1), Q(?B, ?C).
F4(?C, constant) :: F3(?C3)  :- Q(?B, ?C).
F4(?C, constant) :: F4(?C4, constant) :- Q(?B, ?C).
```

Figure 6.7: Shadowing Rules Using Simple Projection

```
F3(?C) :: F3(?C1)            :- S(?A), =(?A, ?v1), F1(?v1, ?B), Q(?B, ?C).
F3(?C) :: F4(?C2, constant)  :- S(?A), =(?A, ?v1), F1(?v1, ?B), Q(?B, ?C).
F1(?v1, ?B) :: F2(?A, ?B, ?v2) :- S(?A), =(?A, ?v1), Q(?B, ?C).
F4(?C, constant) :: F3(?C3)  :- S(?A), =(?A, ?v1), F1(?v1, ?B), Q(?B, ?C).
F4(?C, constant) :: F4(?C4, constant) :- S(?A), =(?A, ?v1), F1(?v1, ?B), Q(?B, ?C).
```

Figure 6.8: Shadowing Rules Using Complete Projection

Definition 39 The **complete projection** of a goal clause $\mathcal{S} = C_1, \ldots, C_m$ relative to a set of variables is defined by $\mathcal{S} \downarrow^*_V = C_{i_1}, \ldots, C_{i_m} (i_1 < i_2 < \ldots < i_m)$ where the literal C_{i_j} appears in $\mathcal{S} \downarrow^*_V$ when $appears(C_{i_j}) \cap V \neq \emptyset$ or there exists a literal term C_{i_k} such that $C_{i_j} \lhd^* C_{i_k}$ and $appears(C_{i_k}) \cap V \neq \emptyset$. ∎

Again using the solution schema $\mathcal{S}_1$, $\mathcal{S}_1 \downarrow^*_{\{?B\}} =$ S(?A), =(?A, ?v1), F1(?v1, ?B), Q(?B, ?C).

The complete projection rather than the simple projection can be used to calculate the constraining clause for the shadowing rule. Figure 6.8 shows the shadowing rules generated for the goal in Figure 6.5 when the complete projection is used.

Using a more complex shadowing rule may result in the selection of fewer goals to re-attempt, but requires more computation to select those goals. This computation is similar to the computation involved in re-attempting the goals. In a sense, an overly-refined shadowing rule shifts computation from attempting goals to the selection of goals to attempt, but may not decrease the overall time spent in consistency maintenance. Second, since the full context of a rule cannot, in general, be recreated, the rule may contain variables that are unbound during the evaluation of the rule. Unbound variables in the constraining clause require increased searching during rule evaluation.

When the target literal matches a fact, checking that the goal to be retried originally searched the datapool in which the source literal is being asserted serves as a fast check to determine whether the goal must actually be retried. This check is a secondary filter on the goals that must be retried. The presence of the secondary filter means that that a simpler shadowing rule that selects more values may be better overall than a more complex rule that selects fewer values.

COLANDER currently generates shadowing rules using the simple projection when the source is either a fact or an entity property. The complete projection is used when the source is a subtree property, since subtree references in the complete projection can be

evaluated relative to the subtree of the source.

6.5.5 Shadowing within a Single Datapool

While the definitions of shadowing account for shadowing between different data values, shadowing within a single datapool of one value by another when both have the same functor name may be unaccounted for. Such shadowing arises when a value can appear multiple times within a single datapool. In some cases, adding a new value will shadow other values having the same functor name; in other cases, no such shadowing occurs. Such shadowing can only occur between two facts in COLANDER since subtree and entity properties are single-valued.

To handle this situation correctly and efficiently requires that the author of a language description explicitly specify the various key fields of a fact. The **key fields**[8] are those arguments that are used to look up values: they correspond to the input arguments in the data literal. Key fields are always bound to ground values when a fact must be matched. If a fact value can appear at most once within a datapool, no key fields need be specified. Two different facts within a single datapool can shadow each other only when their key values are identical.

For instance, suppose "bound(*?name*, *?entity*)" is a fact that represents a binding between a name and an entity, and the same name can be bound to two different entities within the same datapool. Suppose further that this fact is always accessed using the *?name* field as the key. Then two instances of bound can shadow each other within the same datapool only if their *?name* fields match. If the *?name* field is declared to COLANDER as a key field, the following shadowing rule

$$\texttt{bound}(?name, ?entity1) :: \texttt{bound}(?name, ?entity2)$$

can be inferred. Shadowing rules can be generated directly from the specification of key fields.

6.5.6 Maintained Subtree Properties

Maintained subtree properties are values having a procedural definition. During the generation of solution schemata, the appearance of a maintained property can be treated as either a data literal or as the head of a procedure definition. During the expansion of the original goal, maintained properties are treated like data literals. Later, a new goal consisting only of a call to the maintained property is created, and the maintained property is expanded as a procedure. A literal that represents the maintained property is appended to the list of solution schemata generated by this goal so that the shadows analysis will be correct.

For example, suppose that the mytype property of a subtree is defined by

mytype(*?subtree*, *?value*) :- **sometype**(*?subtree*, *?value*).
mytype(*?subtree*, *?value*) :- **another-type**(*?subtree*, *?value*).

[8] This use of "key" differs slightly from the usage found in database terminology. As used here, the key is a lookup key; the values of the key fields may not uniquely determine a value in the database.

When **mytype**(*?subtree*, *?value*) is encountered during the expansion of some other goal, it is treated as a data literal. Later, the goal "← **mytype**(*?subtree*, *?value*)" is expanded into the three solution schemata

$$\begin{aligned} \mathcal{S}_1 &= \textbf{sometype}(\textit{?subtree}, \textit{?value}). \\ \mathcal{S}_2 &= \textbf{another-type}(\textit{?subtree}, \textit{?value}). \\ \mathcal{S}_3 &= \textbf{mytype}(\textit{?subtree}, \textit{?value}) \end{aligned}$$

where $\mathcal{S}_3$ is the specially-added data literal. Shadowing analysis of this set of solutions will create the shadowing rules **sometype**(*?subtree*, *?value*) :: **mytype**(*?subtree*, *?value*) and **another-type**(*?subtree*, *?value*) :: **mytype**(*?subtree*, *?value*).

6.5.7 Improving the Shadowing Rules

This section presents several observations and heuristics that act to improve the set of shadowing rules that are calculated from a language description.

Rule Subsumption

The definitions of shadowing give rise to large numbers of shadowing rules of the form $C_{source} :: C_{target} :- C_1, \ldots, C_m$. Only the most general such rules are of interest in the actual system; any rules that are subsumed by other rules should be eliminated in favor of the general case. Subsumption can be used to eliminate more specific shadowing rules in favor of more general rules. The subsumption algorithm is based on the notion of θ-subsumption [48]:

Definition 40 Let C, D be two clauses and let the notation $||C||$ denote the number of literal terms in C. Then C **θ-subsumes** D if $||C|| \leq ||D||$ and there exists a substitution θ such that $C\theta \subseteq D$. ∎

It is the case that if C θ-subsumes D, then C logically implies D.

Equality of Source and Target

If the source and target are identical (not just unifiable), then the shadows analysis code suppresses that rule. This is just a variant case of rule subsumption. The special case of a fact appearing more than once in a datapool is handled using a separate mechanism.

Bound Collection Heuristic

When a shadows rule is created, it is usually possible to determine which variables appearing in the clause represent datapools, and which of those variables are bound. The datapool into which a source fact is being asserted is always known. An unbound datapool variable appearing in either the target or the constraining clause of the rule forces the evaluator to search *all* of the datapools. One heuristic used in COLANDER is to eliminate any literal from the constraining clause of the shadows rule in which an unbound datapool variable is used. The same heuristic is applied to variables that denote entities or subtrees in properties.

6.6 COLANDER and Shadows Analysis

COLANDER is able to generate shadowing rules based on either the simple projection or the complete projection. Shadowing rules are represented using a standard mechanism of COLANDER: the database trigger mechanism. When a new value is added to the database, the on-assert goals that match that value are executed. Matching and executing a shadows rule may result in other goals being added to the list of unsatisfied goals. Essential to this implementation is that when a fact is added to a datapool, the datapool is known. This datapool, then, helps to determine the datapools that must be searched for values to shadow.

The remainder of this section discusses the interactions between special constructs in COLANDER and the analysis of shadowing.

6.6.1 Dynamic Procedures

Unlike PROLOG, the current implementation of COLANDER does not allow new procedure clauses to be added during the evaluation of a LCG. Two requirements are actually imposed by COLANDER: (i) all procedure clauses must be known when the LCG is analyzed for shadowing and (i) procedure clauses are independent of any specific datapool. Disallowing dynamically-added procedure clauses removes any need to track dependencies between procedure clauses during evaluation.

There are several ways to loosen these restrictions. First, new procedure clauses could be added (dynamically) to the end of existing procedures, but procedure clauses remain independent of specific datapools. Second, procedure clauses could be associated with specific datapools. The order in which datapools are searched would then determine which procedure clauses are actually attempted. Third, procedure clauses could be associated with classes of datapools, either statically or dynamically.

New procedure clauses can introduce new sources of shadowing. Thus, adding a new procedure clause dynamically may require updates to the shadowing rules that have already been calculated. If the system tracks which goals use what procedures (information that is available during shadows analysis), it is straightforward to re-perform shadowing analysis for each goal affected when a procedure definition changes. If new shadowing rules are generated as a result of adding a new procedure clause, the existing data in the database must be checked against the new shadowing rules to ensure that the database is consistent.

A more serious problem, at least computationally, arises when procedure clauses are associated with particular datapools or classes of datapools. In this case, the strict ordering on clauses in a procedure no longer holds—clauses are ordered only within a datapool. Since all possible orders of clauses (or groups of clauses) have to be considered, the derivation of solutions becomes much more expensive computationally. In addition, investigating all possible combinations of clause orderings may infer spurious shadowing that will not occur in practice.

Finally, unless shadowing rules are related directly to the actual procedure clauses that create the shadowing, there would be no way to correctly handle the *deletion* of a procedure clause. Because of subsumption, a given shadowing rule may result from many procedure clauses; the rule cannot be removed unless all of its related procedure clauses are

removed.

6.6.2 Shadowing Analysis for Mapping Functions

Mapping functions specify one or more goals to apply to some other subtree in a syntax tree. Two mapping functions are provided in COLANDER. The first, for-each-child, applies a given iterator goal to each child in a specified list of children, and succeeds if any of the applications of a goal to a child succeeds. The second, for-all-children, is like for-each-child, but succeeds only if the goals applied to all of the specified children succeed. In both for-each-child and for-all-children, the operator of the children must be specified. The syntax of a mapping function is

name(*operator*, *?child*, *child-list*, *iterator-goal*, *result-vars*, *?result*)

Shadowing analysis for mapping functions fits into the normal shadowing analysis with two exceptions. First, when a mapping function is encountered in a normal analysis, the analysis of the goals being mapped is delayed and then handled separately. The terms to the left of those goals are collected into a list of literals so that the context of the goal can be re-established during analysis of the iterator. When the iterator is analyzed, all of the goals in the iterator have to be analyzed as well. To establish the context for *?child*, a call to self-node(*?child*) is prepended to each of the iterator goals.

Second, after posting the information for delayed analysis, the iterator goal is replaced in the solution schema by a special form that binds the result variable and which cannot be the target of a shadowing rule. That it cannot be a target indicates the normal use for the result variable—as a variable dependent only upon the input variables to the iterator and the results of evaluation at the children. (If the expression could be a shadow target, then the consistency manager would have to retain a unique fact for each execution of an iterator, so that its supports information could be retained.)

6.6.3 Bagof and All-Solutions

COLANDER provides two ways to collect all of the results from executing a goal: bagof and all-solutions. The results of the all-solutions primitive are under consistency maintenance. Thus if a new solution could arise, the clause containing the all-solutions must be re-attempted. The evaluation of all-solutions is semantically equivalent to bagof, except that the evaluation of all-solutions causes the results to be stored in the database. These results are represented by a special fact that can be shadowed.

During shadowing analysis, an all-solutions form is treated as the expansion of its goals, except that an extra solution schema including the "all" fact that will represent the results is added to the list of solution schemata. This "all" fact will bind the *?result* and *?result-vars* variables and so will be shadowed by any earlier term that could affect the outcome.

6.6.4 Not and Notever

COLANDER supplies two ways to describe negation: not and notever. The primitive not is the normal PROLOG "not" operator. It is implemented using negation-as-failure. The

literal **not**(*goal*) succeeds only when the *goal* fails. This implies that none of the variables bound within the **not** retain their bindings if the negation succeeds.

Unlike **not**, the results of the **notever** primitive are under consistency maintenance. Thus if a new solution could arise that would cause the **not** to fail, then the clause containing the **notever** will get re-attempted. The evaluation of **notever** is semantically equivalent to **not**, except that the evaluation of **notever** causes the results to be stored in the information repository. These results are represented by a special fact that can be shadowed.

During shadowing analysis, a **not** form is treated specially so that variable bindings will be correctly handled. In particular, no variables will be bound by a literal appearing in the expansion of *goal*. Since the **not** operator succeeds only when its goal fails, the generation of solution schema simply treats **not** operators as primitives that do not bind any unbound variables.

A **notever** literal is treated like a **not** with the same goal, except that a second analysis is performed to expand *goal* in its non-negated form. During the second analysis of *goal*, an extra solution form including a special "never" fact is added to the end of the list of solution schemata. This "never" fact will be shadowed by any earlier term that could affect the outcome of the **notever** literal. It functions much like the "all" fact used to track consistency with **all-solutions** literal—retaining information for later use by the consistency manager.

During the second analysis, the *goal* clause is not treated as a negation, since the question is whether the *goal* will succeed, thereby causing the **notever** to fail, and so the goal that triggered the **notever** in the first place to be attempted again. Thus the second analysis of **notever** is really just like the analysis for **all-solutions**.

6.7 Consistency Maintenance and COLANDER

The consistency maintenance code in COLANDER updates the database to match the current editing state of the document. It does this by tracking the changes to the internal tree that result from incremental parsing.

This section first presents the aspects of a COLANDER description that are crucial to updating the contextual constraints of a document. Then, several of the more complicated portions of the consistency manager are discussed.

6.7.1 COLANDER and Incremental Goal Evaluation

In COLANDER, the goals associated with a structure in the syntax tree are divided into two categories: first-pass goals and second-pass goals. **First-pass goals** can be used to establish the context that is propagated to the substructures of a structure. In general, first-pass goals allocate, initialize, and propagate new context datapools and other collections.

Second-pass goals are the remaining goals associated with a structure in the syntax tree. Second-pass goals can fail when evaluated; if they do fail, they are added to a worklist to be attempted at a later time.

The collections owned by a structure persist even though their contents or their properties may change. Once an collection is created by a structure, it remains with that

structure until the structure is removed from the internal tree. In particular, if a node is split and reused during parsing, the collections associated with that node will persist.

6.7.2 The Goal Update Cycle

Updating the state of the document divides naturally into several phases.

1. Place the goals that were not satisfied prior to this update cycle on a worklist. They may become satisfied during this update cycle.

2. Remove from the database every data value owned by or in any datapool owned by any subtree that was removed from the internal tree during reparsing. When a value $\mathcal{D}$ is removed, add the goals N.$\mathcal{G} \in$ *Justifies*($\mathcal{D}$) to the worklist.

3. Perform a preorder tree walk evaluating the first-pass goals associated with all of the subtrees which received a different context datapool than it received previously. If a subtree receives a new context datapool during this phase, add all of its goals to the worklist. Because context datapools are strictly inherited in COLANDER, this tree walk can stop propagating context datapools whenever a subtree is reached that is unchanged and that inherits the same context datapools as it received during the previous evaluation cycle.

   During this pass, recheck and revise structural subtree properties as necessary.

4. Order and evaluate the goals in the worklist to be attempted[9]. Goals that will assert new facts should be attempted before goals that only use the contents of the database. The left-to-right tree-walk in the evaluation process takes care of this ordering automatically for most programs and programming languages, since it meshes well with a declaration-before-use style of programming. As evaluation proceeds, other values may be added or removed from the database, leading to the reevaluation of goals not initially present on the worklist.

5. For unsatisfied goals, take the necessary actions to report the error.

6.7.3 Subtree Properties

Subtree properties are unique named values associated with subtrees of the internal tree. They come in three varieties: maintained, structural, and extrinsic. Subtree properties are always owned by the subtree for which they encode information.

Maintained subtree properties act like memoized procedures; their values are computed on demand and the value is retained until some change in the database forces reevaluation of the maintained property. Maintained properties cannot be directly evaluated. Every maintained property was originally evaluated on demand, during the satisfaction of some goal. When a maintained property must be retried, COLANDER simply adds the goals that are supported by that property to the worklist.

[9]The prototype evaluator does not reorder the goals on the worklist, with the exception that a subtree appears at most once on the worklist. The undetermined goals of a subtree are evaluated in the order that they appeared in the language description.

Structural subtree properties provide access to information about the (local) structure of the internal tree. As with maintained properties, they are evaluated on demand. Structural properties are automatically revised by the consistency maintenance process. When the value of a structural property changes, the value is removed from the database and any goals supported by that property are added to the worklist.

Extrinsic subtree properties are updated by the consistency manager only if they are asserted by clauses in the COLANDER description.

6.7.4 Evaluation of Shadowing Rules

Shadowing rules are treated identically to database triggers where the constraining clause and the target are concatenated to form the *clauses* of the trigger.

The implementation of on-conditions is careful to check the triggers only when necessary. This means delaying the evaluation of triggers until a value is definitely asserted or retracted.

6.7.5 Mapping Functions

The mapping functions in COLANDER require special handling during consistency maintenance and evaluation. Recall that there are two forms of mapping functions: one that requires successful evaluation of the goal at all of the children, and one that succeeds whenever the goal succeeds at any of the children.

What happens when a mapped goal fails at a child? In this case, the child subtree is tagged with a pointer to the original goal in which the mapping function appeared. When the failed (mapped) goal is retried, the evaluator actually attempts the original (parent) goal for each child. The COLANDER prototype does not attempt to isolate the children for which the mapped goal failed.

6.7.6 Dependencies on the Order of Subtrees in an Internal Tree

COLANDER provides primitive functions to determine the relative ordering of two subtrees within a single syntax tree. In *Pan I*, no special actions have to be taken to track the ordering information. The parsing and node reuse algorithms implemented in COLANDER assure that the relative ordering of subtrees within the internal tree is preserved unless one of the two trees is actually moved using a subtree operation. A subtree move operation in *Pan I* is implemented as a deletion followed by an insertion, so any properties associated with the original subtree will be retracted.

To compare the relative ordering of two subtrees, the subtree references to each tree must be available. Each reference must either be the result of evaluating a structural property, or must be embedded in a value within the database. In either case, the subtree reference must ultimately have been the value of a structural property. When the subtree of the reference is deleted, that structural property will be retracted, as well as the data values dependent upon it. The normal consistency maintenance code handles order-dependencies on subtrees correctly. In an implementation that preserves subtrees during a subtree move operation (as it should), it will be necessary to change the consistency manager to detect when a previously-calculated subtree ordering is changed.

6.8 Data Structures Used by the Consistency Manager

The consistency manager manipulates several data structures during the evaluation of a goal. Some of those structures are persistent, for example, the information associated with each subtree. Other structures are ephemeral, storing information gathered while a single goal is being attempted. Once the goal succeeds (or fails), those data structures are consulted and the appropriate persistent data structures are updated.

6.8.1 Persistent Data Structures

Each subtree in the syntax tree is tagged with information used by the consistency manager. For each goal defined for the subtree, including goals mapped from some other subtree, there is a record that contains either (i) a datapool path indicating that the goal is satisfied, (ii) an error message indicating that the goal failed, or (iii) a marker indicating that the goal is undetermined. Each subtree is also tagged with the collections and values that it owns. To simplify cleaning up the persistent state when a subtree is deleted, a list of the values that support the various goals associated with the subtree can also be kept.

Each value in the database is tagged with the goals that it supports and with the goals that own it. A value in the database also contains information concerning whether the value is actually considered to be present. This information includes the temporary-retraction count.

6.8.2 Data Structures Used During Evaluation

When a goal is being attempted, the consistency manager has to track the values created by the goal, the values used by the goal, and the datapools searched when the goal was attempted. The values used and created are stored in stacks that are may be revised during backtracking. In COLANDER, backtracking through an **assert** of a fact undoes the assertion. However, subtree property values are not removed during backtracking. The list of datapools searched is not revised during backtracking.

If holes are being explicitly represented, then a set of holes used and a stack of holes created are also maintained. The stack of holes created is revised during backtracking, but the set of holes used is not.

When a goal succeeds, the various data structures are traversed, and maintenance actions are undertaken. For instance, the stack of values created is traversed, ownership information is updated, and the on-assert triggers checked. The stack of values used is traversed and the current goal is added to the *Justifies* set for each item. The list of datapools searched is converted into a path stored with the goal.

When a goal fails, the subtree is tagged with the error message describing the failure.

6.9 Tradeoffs between Holes and Shadowing Rules

The tradeoff between holes and shadowing rules remains to be discussed. The strategy of using holes seems most applicable to applications in which the required infor-

mation is almost always present, while the strategy of using shadowing rules is best suited to applications in which the data is sparse. Can this observation be used to improve the performance of a LCG?

The distinctions between classes of data values—facts, subtree properties, and entity properties—provide one basis for choosing when to use holes and when to use shadowing rules. Structural subtree properties are the most "dense"; they always have a value and so never require a hole. For structural properties, then, using holes is optimal. In practice, other subtree properties appear to be fairly dense as well. Facts, on the other hand, are likely to be sparse. So facts go well with shadowing rules. Entity properties seem to be somewhere between.

COLANDER has three configurations that allow experimentation with which strategies to use—always use holes, always use shadowing rules, and a hybrid strategy. The default choice is to use holes with subtree and entity properties, and shadowing rules with facts.

Whether a more detailed analysis of a description could automatically choose to between holes or shadowing rules using a finer discrimination is an open research question. Examination of the shadowing rules could provide one method for choosing: if a data literal is the source for many shadowing rules, it might be most efficient (computationally) to use facts to represent its absence. Tracking the appearance of data literals in the solutions generated during shadowing analysis might also provide useful heuristics. A third approach is to base the choice of holes versus shadowing on performance data gathered while using the LCG to analyze representative programs or documents.

Strategies that rely more on recomputation than consistency maintenance also deserve further study. For instance, with a fast enough static-semantic analyzer, it might be worthwhile to recompute all intraprocedural semantic information while incrementally maintaining interprocedural data.

Chapter 7

Using Colander: Examples and Experience

The theory underlying consistency maintenance in COLANDER is all very nice, but how is the language actually used? This chapter presents excerpts from a COLANDER description of the programming language Modula-2 [104, 140]. The full description appears as Appendix C[1]. Modula-2 was chosen for this example because of the richness of its type system and the complexity of its scope and visibility rules. To simplify the presentation of this material, the reader is assumed to be familiar with programming language concepts in general, and with Modula-2 in particular.

7.1 Data Representations

When writing a COLANDER description, careful attention must be paid to the representation of data used by the description. In the description presented below, linguistic objects such as constants, variables, types, procedures, or modules are represented by entities. Every entity has a *type* and a *mode*. The type of an entity usually corresponds to its declared type in the program. For all linguistic objects except types, the type of the object is itself represented by an entity. The mode of the entity denotes the kind of linguistic object being represented. For instance, the mode of an entity that represents a declared type is the (COLANDER) constant "type". Table 7.1 summarizes the type and mode information used in the description. Expressions and constant expressions are not explicitly represented as entities. Instead, their types and modes are represented using maintained properties of the tree nodes that represent the expression.

A ⟨*constant-descriptor*⟩ is a COLANDER structure. Each constant descriptor is typed, and contains the value of the constant being represented. Table 7.2 shows the constant descriptors used in the description. The structures chosen to represent constants are relatively arbitrary. Since a constant descriptor appears as a mode and is paired with a type descriptor, embedding the type of the constant is redundant. The manner in which the type is embedded is also a matter of choice. Thus, integer constants could be repre-

[1] A smaller example, *Tiny*, is discussed in Chapter 5. The full description of *Tiny* appears in Appendix B.

Object	Mode	Type
Constant	⟨*constant-descriptor*⟩	entity representing the type
Function	function	entity representing the type
Module	module	entity representing the type
Procedure	procedure	entity representing the type
Type	type	⟨*type-descriptor*⟩
Variable	variable	entity representing the type

Table 7.1: Modes and Types of Linguistic Objects

bool-constant(value)	Structure for boolean constants
enum-constant(value)	Structure for enumeration constants
int-constant(value)	Structure for integer constants
ptr-constant(value)	Structure for pointer constants
real-constant(value)	Structure for real constants
string-constant(value)	Structure for string constants

Table 7.2: Structures that Represent Constants

sented using the structure "constant(value)" or "constant(value, INTEGER)" rather than as "int-constant(value)".

A ⟨*type-descriptor*⟩ is also a COLANDER structure. Type descriptors are often composed of references to other types and their descriptors. For instance, the type descriptor for a subrange refers to the base type of the subrange, e.g., a subrange of integer will contain the entity that represents the standard type INTEGER. Table 7.3 summarizes the type descriptors used in the description of Modula-2. The descriptor for a record type contains the datapool in which the facts representing fields of the record are placed. The descriptor for an enumeration type contains a list of the names of the type's constants. Most of the other descriptors contain entities for the types underlying the type being defined.

The fact "bound(name, entity, text-loc)" is used to represent the binding between a name and the linguistic object represented by an entity. "Text-loc" designates the position in the program at which the binding first appears. It is used for checking the requirement that (most) type names be defined before their use. Positions in the text are represented using node references.

Table 7.4 describes the COLANDER facts that are used to represent information about the importation or exportation of names.

7.2 Contexts and Contours

COLANDER provides facilities to cleanly separate the context in which a goal is attempted from the contour that represents the local scope in the program. Contexts are

array-type(list-of-component-type-entities)
enumeration-type(list-of-constant-names)
function-type(list-of-argument-type-entities, result-type-entity)
module-type()
named-type(type-entity)
opaque-type(id-name)
pointer-type(type-entity)
procedure-type(list-of-argument-type-entities)
record-type (datapool)
set-type(type-entity, low, high)
subrange-type (base-type-entity, low-bound, high-bound)

Table 7.3: Type Descriptors

bound(name, entity, text-loc)
imported-name(name, entity, text-loc)
export-unqualified(name, entity, text-loc)
export-qualified(name, entity, text-loc)

Table 7.4: Facts for Representing Bindings, Imports, and Exports

represented by context datapools. The context datapool is a property of the structure of the tree.

Every context datapool in the Modula-2 description contains a fact "contour(dp)" in which "dp" is the local contour. A contour contains the declarations and other data local to a scope in the program. Besides the local contour, a context datapool may contain other facts being used to pass local contextual information down the tree. The idiom

... context(*?env*), <contour(*?contour*), *?env*> ...

is used to bind *?env* to a subtree's context datapool and then to bind *?contour* to the local contour.

Each contour C is characterized by one or more facts asserted in C. The facts used to characterize contours are shown in Table 7.5.

For instance, every contour that contains the imports, exports, and declarations of a module also contains a fact of the form "module-contour(entity, type)" where value of "entity" is the entity that represents the module. Thus a module contour contains a pointer to the module entity with which it is associated. The "type" indicates the type of the module: one of "definition", "implementation", "local", or "program".

Every module is represented by an entity. Every entity representing a module has a "defn-contour" property. If it is an implementation module, it also has a "impl-contour" property associated with it.

module-contour(entity, type)	Present in contours representing module scopes
record-contour()	Present if contour represents a record scope
procedure-contour()	Present in contours representing procedure or function scopes
open-contour()	Present if the contour represents an open scope
parent-contour(?env)	Pointer to parent contour

Table 7.5: Facts that Characterize Contours

```
<lookup(?fact, ?where-found), ?where-found> :-  <?fact, ?where-found>,   !.

<lookup(?fact, ?where-found), ?contour> :-
        <open-contour(), ?contour>,
        <parent-contour(?parent), ?contour>,
        <lookup(?fact, ?where-found), ?parent>.
```

Figure 7.1: Visibility Rules for Pascal

7.3 Visibility Rules

The visibility rules of Modula-2 are complex in comparison to block-structured languages like Pascal. To provide a fuller appreciation for COLANDER, this section briefly presents the Pascal scope rules before presenting to those rules used in Modula-2.

Figure 7.1 shows the COLANDER scope rules for Pascal. A typical use of this rule is given by

<lookup(bound(*?name*, *?entity*, *?text-loc*)), *?contour*>

To map from a name to an entity, "lookup" would be given a literal term "bound(*?name*, *?entity*, *?text-loc*)" with *?name* bound to the name being handled. The procedure "lookup" starts from the given contour and attempts to match *?fact* relative to that contour. If it succeeds, it continues the computation. The cut operator ("!") in the first clause of lookup ensures that if the remainder of the goal fails, the interpreter will not backtrack over the first literal and attempt to search the enclosing scopes. If the attempt in the current contour fails, and if there is a surrounding contour, "lookup" recursively calls itself relative to the surrounding contour. If "lookup" succeeds, then the variables *?entity* and *?text-loc* will be bound to appropriate values.

The scoping rules for Modula-2, shown in Figure 7.2, are much more complex. Modula-2 supports both open (procedure, function, with-clause, and for-loop) scopes and closed (module) scopes. A name is visible within a module scope if it is declared within the module, if it is imported from some other module, or if it is exported from a local (sub) module. Names imported from another module must be explicitly exported from that module. Exports come in two varieties: qualified and unqualified. A qualified export requires that both the module name and the exported name be specified. An unqualified export allows the exported name to be referenced without also specifying the module name.

```
<name-is-imported(?name, ?entity, ?text-loc), ?contour> :-
        <imported-name(?name, ?entity, ?text-loc), ?contour>.

<name-is-imported(?name, ?entity, ?text-loc), ?contour> :-
        <imported-name(?mod-name, ?mod-entity, ??), ?contour>,
        type-descriptor(?mod-?entity, module),
        module-contour(?mod-entity, ?mod-con),
        <name-is-exported(?name, ?entity, ?text-loc, ?module-name), ?mod-con>.

<name-is-exported(?name, ?entity, ?text-loc, ?mod-name), ?contour> :-
        <export-unqualified(?name, ?entity, ?text-loc), ?contour>.

<name-is-exported(?name, ?entity, ?text-loc, ?mod-name), ?contour> :-
        <export-unqualified(?mod-name ??, ??), ?contour>,
        <export-qualified(?name, ?entity, ?text-loc), ?contour>.


<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        <bound(?name, ?entity, ?text-loc), ?contour>.

<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        open-contour(?contour),!,
        <parent-contour(?parent), ?contour>,
        <resolve-binding(?name, ?entity, ?text-loc), ?parent>.

<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        <module-contour(?mod-e, ?mod-type), ?contour>,
        <name-is-imported(?name, ?entity, ?text-loc), ?contour>.

<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        <module-contour(?mod-entity, implementation), ?contour>,
        defn-contour(?mod-entity, ?def-con),
        <resolve-binding(?name, ?entity, ?text-loc), ?def-con>.

<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        global-dp(?dp),
        <bound(?name, ?entity, ?text-loc, ?dp),  ?dp>.
```

Figure 7.2: Modula-2 Visibility Rules

```
/* Definition of type CARDINAL */
:- bound("INTEGER", ?int-e, ??),
   type(?int-e, subrange(??,?low,?high)),
   new-anon-entity(?e,  ( mode(?e, type),
                          type(?e, subrange-type(?int-e, 0, ?high)) )),
   assert(bound( "CARDINAL", ?e, PREDEFINED)).
```

Figure 7.3: Declaring a Primitive Type

The primary scoping procedure in Figure 7.2 is called "resolve-bindings". It uses the helper procedure "name-is-imported". A name is imported (i) if it is directly imported, (ii) if a module from which it has been exported has been imported, or (iii) if the current module is an implementation module and the name was declared in the related definition module. These rules form the three clauses of "name-is-imported". Note that through backtracking search, all of the modules imported into another module will be searched (if necessary). Because of the strong constraints on imports and exports, the order in which the modules are searched is irrelevant. In a language for which the order of appearance of imports is relevant, the both the search rule and the facts used to represent the name space of the program would have to be more complex.

Finally, the Modula-2 scope rule can be defined. It has five clauses:

1. A name is visible in a contour if it is declared there.
2. A name is visible in an open contour if it is visible in the surrounding contour.
3. A name is visible in a module contour if it has been imported into that contour.
4. A name is visible in the contour of an implementation module if it is visible in the contour of the corresponding definition module.
5. Pervasive definitions are visible in all modules.

"Resolve-binding" is a straightforward encoding of these cases.

7.4 Declarations

Figures 7.3–7.5 provide three examples of declarations. The first example is the (axiomatic) goal that establishes the default binding for the standard type CARDINAL. The binding is established in COLANDER's global datapool. Recall that this datapool is searched in the last clause of "resolve-bindings". Since the goal is outermost, it will be executed at language-definition time. The first thing that the goal does is to find the entity *?int-e* that represents the type INTEGER. The standard type INTEGER is itself represented by a subrange type that includes the minimum and maximum integer values of the implementation The goal then allocates a new entity having mode "type" and type "subrange(*?int-e*, 0, *?high*)" where *?high* is the maximum integer value obtained from the definition of INTEGER. Finally, the goal establishes the new binding by asserting the fact "bound("CARDINAL", *?e*, PREDEFINED)" into the global datapool.

```
type_declaration
        =       id "=" type ";" => type_declaration {
        entity-name(type-entity).
        // First Pass
        // Second Pass
         :-
            context(?env),
            pname($id, ?name),
            expr-type($type, ?expr-type),
            <contour(?contour), ?env>,
            <declarable(?name), ?contour> :
                ( "~a is already bound", ?name),
            new-entity(type-entity, ?e,
                        ( mode-of(?e, type),
                          owner(?e, $type_declaration),
                          type-of(?e, ?expr-type))),
            assert(bound(?name, ?e, $type_declaration), ?contour).

        /* Second goal checks for redefinition between definition and implementation module */

        :-  context(?env),
            <contour(?contour), ?env>,
            <contour-type(module(?mod-ent, implementation)), ?contour>,
            defn-contour(?mod-ent, ?def-con),
            type-entity(?type-ent),
            pname($id, ?name),
            <bound(?name, ?def-entity, ?text-loc), ?def-con>,
            valid-type-redefinition(?name, ?type-ent, ?def-entity).
}
```

Figure 7.4: Declaring a Defined Type

Figure 7.4 and Figure 7.5 show the declaration of a named type and of a variable. In the declaration of a type, "expr-type" is a maintained subtree property. Its value is the entity representing the type. Note that the goal checks to see that the name can be declared before it attempts to declare it. Recall that the information following a colon designates an error message to be used when the literal fails. Finally, a second goal ensures that a type declaration within an implementation module is consistent with the declaration given in the definition module.

The declaration of a list of variables in Figure 7.5 uses a mapping function to declare each of the identifiers in turn. Otherwise, it is similar to the declaration of a type.

Figure 7.6 shows the operator definition for a qualified identifier. Qualified identifiers are used to declare types. They represent an access path to a type entity. The objects on the path can be either modules or records; the "." operator is used to separate the elements on the path. Finding the entity designated by a qualified identifier is more complicated than a simple lookup.

Each partial prefix of a qualified identifier must identify either a record or a module. So each identifier in a partial prefix must have an associated contour. All that the large

```
var_declaration
        =       id_list ":" type ";" => var_declaration {
        // Context
        // Constraints
          :-  context(?env),
              <contour(?contour), ?env>,
              expr-type($type, ?expr-type),
              children($id_list, ?children),
              for-each-child(id, ?node, ?children,
                        { :all =>  pname(?node, ?name),

                                <declarable(?name), ?contour> :
                                        ( "~a is already bound here", ?name),
                                new-entity(id-entity, ?e, ( mode-of(?e, variable),
                                                        type-of(?e, ?expr-type))),
                                assert(bound(?name, ?e, ?node), ?contour)
                        }, ??, ??).

        :-  context(?env),
            <contour(?contour), ?env>,
            <contour-type(module(?mod-ent, implementation)), ?contour>,
            defn-contour(?mod-ent, ?def-con),
            children($id_list, ?children),
            for-each-child(id, ?node, ?children,
                { :all =>
                        id-entity(?var-ent),
                        pname(?node, ?name),
                         <bound(?name, ?def-entity, ?text-loc), ?def-con>,
                        valid-redefinition(?name, ?var-ent, ?def-entity)
                        }, ??, ??).
        }
```

Figure 7.5: Declaring Variables

mapping function does is to locate, for each identifier on the path (unless it is the last) the contour associated with that identifier. Once found, a node property "lpool" whose value is the contour found is asserted. The path elements are processed left-to-right. The first element searches the current contour for the binding of the initial identifiers; the other elements use the contour designated by the "lpool" property of their left sibling.

The result computed by the mapping function is a list of the entities corresponding to the successive prefixes of the identifier list. The last entity on the list is the one that is designated by the entire path. This entity is used to define the value of the maintained property "qual-ident-entity".

The definition for qualified identifiers also includes two local procedures as shown in Figure 7.7. The procedure "selectable-contour(*?entity*, *?contour*)" succeeds and binds *?contour* to a contour (if it exists) whenever *?entity* represents a record or a module. "Export-check(*?name*, *?contour*)" succeeds whenever *?name* is visible from outside of *?contour*.

```
qual_id =       id + "." => qual_id {
        qual-id-entity($qual_id, ?ent) :-
            context(?env),
            <contour(?contour), ?env>,
            children($qual_id, ?kids),
            for-all-children(id, ?child, ?kids,
                { :singleton =>
                        pname(?child, ?name),
                        <resolve-binding(?name, ?ent, ?text-loc), ?contour>:
                            ("~a is already bound in this scope!", ?name).}
                { :first =>       /* first looks in contour */
                        pname(?child, ?name),
                        <resolve-binding(?name, ?entity, ?text-loc), ?contour>:
                            ("~a is not bound in this scope!", ?name),
                        selectable-contour(?entity, ?dot-contour),
                        assert(lpool(?child, ?dot-contour)). }
                { :middle =>      /* middle uses lpool */
                        pname(?child, ?name),
                        left-sibling(?child, ?left-sib),
                        lpool(?left-sib, ?inh-contour),
                        <bound(?name, ?entity, ?text-loc), ?inh-contour>:
                            ("~a is not bound in this scope!", ?name),
                        export-check(?name, ?inh-contour),
                        selectable-contour(?entity, ?dot-contour),
                        assert(lpool(?child, ?dot-contour)). }
                { :last =>        /* last uses lpool */
                        pname(?child, ?name),
                        left-sibling(?child, ?left-sib),
                        lpool(?left-sib, ?inh-contour),
                        export-check(?name, ?inh-contour),
                        <bound(?name, ?entity, ?text-loc), ?inh-contour>:
                            ("~a is not bound in this scope!", ?name).
                }, ?ent, ?result),
                last(?result, ?ent).
```

Figure 7.6: Handling Qualified Identifiers

```
selectable-contour(?name, ?entity, ?out-contour) :-
                type-descriptor(?entity, record(?contour)).
selectable-contour(?name, ?entity, ?out-contour) :-
                mode-of(?entity, module),
                defn-module(?entity, ?out-contour).

<export-check(?name), ?contour> :-
                <record-contour(), ?contour>, !.
<export-check(?name), ?contour> :-
                <contour-type(module(??, ??)), ?contour>,
                <export-qualified(?name, ??, ??), ?contour>.
<export-check(?name), ?contour> :-
                <contour-type(module(??, ??)), ?contour>,
                <export-unqualified(?name, ??, ??), ?contour>.
```

Figure 7.7: Local Procedures for Qualified Identifiers

```
same-type(?t1, ?t1).
same-type(?t1, ?t2) :- type-descriptor(?t1, named-type(?t2)).
```

Figure 7.8: Type Equality in Modula-2

7.5 Type Checking

Type checking in the Modula-2 description is implemented using several axiomatic procedures along with two lookup tables. The axiomatic procedures define what it means for two types to be the same, what it means for two expressions to be compatible, and what it means for an expression to be assignable to a variable. The tables encode the information about the various unary and binary operators: what types they require and what types they return. Only the procedure for determining type-equality is presented here.

According to Pomberger [104]:

> "Two objects a (of type $t1$) and b (of type $t2$) are of the *same data type* if one of the following conditions hold:
>
> 1. a and b are variables (or formal parameters) which occur in the same identifier list in a declaration, for example ....
> 2. a and b are enumeration constants of the same enumeration type.
> 3. $t1$ and $t2$ are given by the same name.
> 4. $t1$ and $t2$ are given by type names and the name of one is used for the name of the other, for instance ...."

The clauses in the procedure "same-type" of Figure 7.8 define equality of types in this description.

The data representation for enumeration constant names ensures that two enumeration constants defined in the same enumeration will have the same associated type entity. Similarly, two identifiers declared in the same declaration will have the same associated

Number of operators	154
Number of operators having semantic actions	116
Number of facts declared	27
Number of maintained properties declared	11
Number of entity properties declared	6
Number of operators that allocate entities	24
Number of operators that allocate contexts	14
Number of global procedures	146
Number of maintained property procedures	77
Number of outermost goals	9
Number of first-pass goals	15
Number of second-pass goals	57
Average number of cases per procedure	2.08
Average number of schemata in a solution schema	7.9
Number of schemata generated	2076
Number of schemata that 2-loop	503
Number of possible shadowing rules investigated	1767
Number of shadowing rules generated	66
Number of data literals having shadowing rules	22

Table 7.6: Summary of Modula-2 Description

type entity. The first clause of "same-type" therefore handles cases (1)–(3) of the given definition. The other clause of "same-type" handles equivalence between named types.

7.6 The Modula-2 Description as a Whole

The entire COLANDER description of the static-semantic constraints of Modula-2 is provided in Appendix C. Among the language issues present in the full language description but not discussed here are: sets and set constants, records, with statements, procedure and function definition, and type checking of expressions. Table 7.6 summarizes various static measurements of the Modula-2 description. Overall, 66 shadowing rules were created, for 22 different data literals. Only 17 of the shadowing rules are used when holes are used for property values.

The Modula-2 description took roughly 8 person-weeks to produce from a given Modula-2 syntax. This time included the time needed for a novice Modula-2 programmer to understand the language. In contrast, the description for *Tiny* required less that a day; recently, a programmer new to using COLANDER added typed numeric constants to *Tiny* in a short afternoon. As experience accrues, the time required to create a new language description is expected to decrease.

Figure 7.9 contains a single, small, Modula-2 program. The program is comprised of two modules, `IO` and `Factorial`. Two procedures, "ReadCard" and "WriteCard" are de-

```
DEFINITION MODULE IO;
        EXPORT WriteCard, ReadCard;

        PROCEDURE WriteCard(x : CARDINAL);
        PROCEDURE ReadCard(VAR num : CARDINAL);
END IO.

(* Modula2 Program to compute the factorial function *)
MODULE Factorial;
FROM IO IMPORT ReadCard, WriteCard;

VAR
        X,      (* input *)
        Fact,   (* result *)
        N       (* iteration counter *)
        : INTEGER;

BEGIN (* factorial *)
        ReadCard(X);
        Fact := 1;
        N := 1;
        IF ( X <> 0 )
                THEN
                WHILE (N <> X) DO
                        N := N + 1;
                        Fact := Fact * N;
                        END;
                END;
        WriteCard(Fact);
END Factorial.
```

Figure 7.9: A small Modula-2 Program

fined in and exported from IO. The program module Factorial imports those procedures from IO and computes a simple arithmetic function. The program, while small, is quite illustrative. It contains imports, exports, multiple modules, declarations, simple type-checking, type-coercions, and procedures. In all, 12 different identifiers appear, there are 2 modules, 2 procedures (plus the body of Factorial), 5 declarations, and 7 statements.

To perform semantic checking (excluding parsing) on the 27-line example program in Figure 7.9 required an average of 650 msec of CPU time[2] on a lightly-loaded SUN 3/60. This corresponds to almost 2500 lines/CPU minute. Other average CPU times on the same file: 78 msec to delete an identifier such as "Fact" or "N"; 50 msec to add the same identifier; 100 msec to delete an exported identifier used within another module, such as "ReadCard" in Factorial; 120 msec to add the same identifier after it was removed; and 50 msec to add

[2]The times given exclude the time spent in garbage collection.

Total number of facts	114
Number of facts in global datapool	69
Total number of holes	35
Number of datapools	23
Number of entities	17
Number of entity property values	40
Average number of properties per entity	2.35
Number of subtrees (including lexemes) having properties	121

Table 7.7: Database Statistics

Average number of facts per datapool	2.04
Average number of holes per datapool	1.59

Table 7.8: Averages for Datapools, Excluding the Global Datapool

Average number of goals supported by a fact	1.7
Average number of goals supported by an entity property	0.75
Average number of goals supported by a subtree property	1.12

Table 7.9: Supports Information

a new identifier in the `Factorial` module that shadowed one of the imported identifiers (such as "ReadCard").

Table 7.7 presents information gathered from COLANDER when this program was analyzed. In Table 7.7, the number of holes represents the total number of holes created when no shadowing rules were used. The facts in the global datapool represent data that is independent of any program.

Tables 7.8–7.9 contain statistics on the storage used by consistency maintenance.

Chapter 8

Conclusion

The prototype implementations of *Pan I*, LADLE, and COLANDER show that the approaches adopted in this research are viable. In general, the desiderata set forth on page 3 have been met:

- The database orientation of COLANDER allows the information gathered by COLANDER to be shared among tools. Currently, *Pan I* is being augmented with a tree-oriented viewer and with semantic-based viewing facilities, both of which use the information gathered and maintained by COLANDER.
- The complete description of Modula-2 shows that well-known and complex languages can be supported. Descriptions of "C" and the COLANDER language are under development.
- The user interface of *Pan I* varies little among languages. In fact, most languages have identical user interfaces except for their operand hierarchy. Experience with the operand hierarchy indicates that while it contributes an important dimension to editing, most editing interactions remain text-oriented. This, of course, may change as *Pan I* widens its base of users.
- *Pan I* is able to support both text- and structure-oriented editing without placing undue restrictions on the user.
- Shielding the user from the details of the internal representation of documents has proved more difficult than originally anticipated, although the operand hierarchy mechanism helps in selection and navigation around trees. Of course, removing the standard tree navigation commands from the default bindings would go a long way in hiding the actual internal structure, and would force reliance on the operand hierarchy!
- *Pan I* provides error detection and recovery. Syntactic error detection is handled by the underlying parser, CAMEL, developed at UC, Berkeley. CAMEL implements a powerful panic-mode error recovery system [29]. The error recovery mechanism is table-driven. A complete syntax description includes the recovery declarations as well as other LADLE definitions. Semantic error recovery is integrated with the consistency maintenance component of COLANDER.

- As the Modula-2 description of Appendix C shows, a COLANDER description is powerful, while remaining reasonably readable. Much of the complexity of the description inheres in the complexity of Modula-2 itself.

- Finally, *Pan I* has been in use as a text editor for more than three years. Its mouse and region-oriented interface makes it an easy editor to use. Recently, a directory editor and a reference manual browser have been added, increasing its usefulness as an editor. The syntactic analysis portions of *Pan I* have been available for over two years. The semantic-checking portions of *Pan I*, being a relatively new prototype, will benefit by performance improvements.

The *Pan* group at Berkeley is already developing new features based on COLANDER. In this section, we briefly suggest some other areas that deserve attention.

Pan II, based on the lessons of *Pan I*, is already under development. In *Pan II*, the close reliance on textual representations present in *Pan I* will be removed. Additionally, *Pan II* will integrate other forms of editing beyond text- and structure-oriented, and will include much of the functionality present in the VORTEX system [24].

The theory of grammatical abstraction, developed originally for LADLE, has recently been revised [23]. Besides that rework, one area for further exploration is to look into allowing more complex tree transformations between the concrete and abstract syntax trees.

The prototype implementation COLANDER focussed on developing a test bed to experiment with logical constraint grammars and language descriptions. As discussed above, our initial experience shows the promise of these techniques. Areas for further research in order to improve COLANDER are discussed in the following paragraphs.

Analyzing Data Dependencies: The prototype implementation of COLANDER makes no concerted attempt to minimize visits to nodes in the internal tree. While most of the interesting interactions between data values occur through the database, a complete description of a language also shows many local dependencies among subtree properties. Better analysis of data dependencies and the ordering evaluation in a logical constraint grammar might improve the overall performance of the consistency manager.

Integrating COLANDER with a Program Database: A primary tenet of this research is that a program database is required for the full benefits of a language-based editor to accrue. The COLANDER prototype includes a very simple database facility. Integrating the consistency maintenance aspects of a logical constraint grammar with a more complete and powerful program database or knowledge base would result in a better system and is needed for long-term experimentation.

Improving Performance: Improved dependence analysis and a better data manager are two aspects of improving the performance of COLANDER. Hard performance data on COLANDER is scarce, for several reasons. First, the development of COLANDER initially focussed on a complete, robust implementation that can support language descriptions as large as Modula-2's. Developing the Modula-2 description itself required substantial effort. Second, without a program database, much of the time spent in analyzing

a program or document is spent in the initial analysis. Persistent storage for data will amortize that expense. Presently, the cost of initial analysis is paid each time the editor visits a file for the first time. Finally, good performance models for evaluating language-based editing systems must be developed. What is an accurate (or reasonable) model for performance evaluation in an interactive setting, where the language, the language description, the document, and the editing style of the user can each affect overall system performance? This is an important area for further research.

In terms of the actual implementation, some areas for investigation include the adaptation of standard PROLOG compilation and optimization techniques and an improved interpreter.

Support for Developing Language Descriptions: Language descriptions for simple languages appear quite tractable, but simple languages are quite rare. The description of Modula-2 is lengthy, largely because of the size and complexity of the language. Writing the Modula-2 description was a lengthy process as well, partly because the description and the techniques used in the description evolved together.

Logical constraint grammars impose new styles of programming and data representation. Since relatively few languages have been fully described, the process of writing new descriptions remains stochastic. The process of developing new language descriptions would be enhanced by a better environment for debugging a description, as well as by the standardization of techniques for representing data within a language description. Enlarging the collection of language descriptions may also lead to relatively standard packages that describe various aspects of programming languages. Such packages might include visibility rules, type-checking and type inference, call-graph manipulations, and data representations.

The description of Modula-2 contains portions that are quite regular. This suggests that some higher-level descriptive techniques based on LCGs are possible.

Trading Computation for Storage: How applicable is consistency maintenance when machine cycles are cheap? Why maintain lots of information when it is so easy to recompute it? Like the tradeoffs between using holes and shadowing rules, there are tradeoffs between how much information to maintain and how many cycles are available for recomputation. The prototype application developed during this research maintains dependency information to a very fine grain. However, the techniques used can be adapted to hybrid systems. For instance, in performing static-semantic checking, a system might choose to maintain only interprocedural data and to recompute all intraprocedural data. In this strategy, a procedure would be the smallest unit to be checked. Simple recomputation strategies, however, do not eliminate the need for fine-grained analysis when other forms of textual analysis such as semantics-directed viewing or Masterscope-like analysis are implemented using LCGs.

Logical constraint grammars also provide a basis for several areas of further exploratory research. The next paragraphs suggest several directions for exploratory research dealing with logical constraint grammars.

Integrating COLANDER with Natural Semantics: COLANDER is a high-level language for language descriptions. However, it is possible to regard the entire COLANDER system as a lower-level execution vehicle for a much higher level descriptive technique. One such technique is natural semantics [66]. Attali [10] transforms a TYPOL description into an attribute grammar and then evaluated using standard attribute evaluation techniques. Instead, a TYPOL description could be compiled directly to a logical constraint grammar.

Type Inference and Polymorphism: Type inference and polymorphism, as found in functional languages like ML [53] and Haskell [60], can be implemented using LCGs. This application is a natural for a logic-based programming language, since type inference algorithms rely on unification for solving systems of type equations. It should also be possible to add type inference to language descriptions for languages such as Modula-2 that nominally require fully-specified type declarations. In this case, the editing system could supply declarations that were omitted by the user of the editor.

Adding Events: At present, constraints of a logical constraint grammar are evaluated only in response to changes in the editing environment. A more generalized event model could be added to the logical constraint grammar paradigm. (This extension is suggested by the Kaiser's approach to adding dynamic semantics to attribute grammars [68].) By affiliating constraints with events, much more powerful and dynamic systems could be implemented.

Implementing New Applications: As noted, the facilities of *Pan I* and COLANDER can support the creation of textual analysis tools like Masterscope [86] or Microscope [7, 79]. Other applications already being considered include a more powerful and general approach to viewing documents [132] and incremental compilation.

Experience has shown the power and the flexibility of *Pan*, grammatical abstraction, and logical constraint grammars. By using the capabilities already provided, developers can implement new facilities not normally encountered in multi-language environments. By extending these models, one can provide more powerful and more complete front-ends for development environments.

Bibliography

[1] ACM. *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, Oregon, June 8–10, 1981. Appeared as Sigplan Notices, 16(6), June 1981.

[2] ACM. *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, Williamsburg, VA, January 26-28, 1981.

[3] ACM. *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 25–27, 1982.

[4] ACM. *Proc. of the SIGPLAN '86 Symposium on Language Issues in Programming Environments*, Seattle, Washington, June 25–28, 1985. Appeared as Sigplan Notices, 20(7), July 1985.

[5] A. V. Aho and S. C. Johnson. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, August 1975.

[6] C. N. Alberga, A. L. Brown, Jr. G. B. Leeman, M. Mikelsons, and M. N. Wegman. A program development tool. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages* [2].

[7] James Ambras and Vicky O'Day. Microscope: A knowledge-based programming environment. *IEEE Software*, 5(3):50–58, May 1988.

[8] J. Archer. COPE: a cooperative programming environment. TR 81–459, Department of Computer Science, Cornell University, Ithaca, June 1981.

[9] J. Archer. The design and implementation of a cooperative program development environment. TR 81–468, Department of Computer Science, Cornell University, Ithaca, January 1981.

[10] I. Attali and P. Franchi-Zannettacci. Unification-free execution of TYPOL programs by semantic attribute evaluation. In Kowalski and Bowen [77], pages 160–177.

[11] Isabelle Attali. Compiling TYPOL with attribute grammars. In P. Deransart, B. Lorho, and J. Maluszński, editors, *Programming Languages Implementation and Logic Programming*, number 348 in Lecture Notes in Computer Science, pages 252–272, Berlin, Heidelberg, New York, 1988. Springer-Verlag.

[12] Hamid Bacha. MetaProlog design and implementation. In Kowalski and Bowen [77], pages 1371–1387.

[13] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Trans. on Programming Languages and Systems*, 8(4):547–576, 1986.

[14] Robert A. Ballance, Jacob Butcher, and Susan L. Graham. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[15] Robert A. Ballance and Michael L. Van De Vanter. Pan I: n introduction for users. Technical Report 88/410, Computer Science Division, UC Berkeley, March 1988.

[16] Robert A. Ballance, Michael L. Van De Vanter, and Susan L. Graham. The architecture of Pan I. Technical Report 88/409, Computer Science Division, UC Berkeley, March 1988.

[17] George McArthur Beshers and Roy Harold Campbell. Maintained and constructor attributes. In *Proc. of the SIGPLAN '86 Symposium on Language Issues in Programming Environments* [4], pages 34–42. Appeared as Sigplan Notices, 20(7), July 1985.

[18] Christina L. Black. Ppp: A pretty-printer for Pan. Master's thesis, Computer Science Division—EECS, University of California, Berkeley, California, 94720, In preparation.

[19] Alan Borning and Robert Duisberg. Constraint-based tools for user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[20] P Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In Henderson [56], pages 14–24.

[21] Kenneth A. Bowen and Tobias Weinberg. A meta-level extension of Prolog. In *IEEE Symposium on Logic Programming 1985*, pages 48–53, 1985.

[22] Frank J. Budinsky, Richard. C. Holt, and Safwat G. Zaky. SRE—a syntax-recognizing editor. *Software—Practice & Experience*, 15(5):489–497, May 1985.

[23] Jacob Butcher. Ladle. Master's thesis, Computer Science Division—EECS, University of California, Berkeley, California, 94720, 1989.

[24] Pehong Chen, J. Coker, Michael A. Harrison, J. McCarr ell, and S. Procter. The VorTeX Document Preparation Environment. In J. Desarménien, editor, *TEX for Scientific Documentation, Proc. 2nd European Conf. Strasbourg, France. June 19-21*, 1986. Lecture Notes in Computer Science, No. 236, pp. 45-54 (1986).

[25] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, third, revised and extended edition, 1987.

[26] E. Cohen. Text-oriented structure commands for structure editors. *SIGPLAN Notices*, 17(11):45–49, 1982.

[27] Reidar Conradi, Tor M. Didriksen, and Dag Wanvik, editors. *Advanced Programming Environments*, number 244 in Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 1986. Springer-Verlag.

[28] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[29] Robert Paul Corbett. *Static Semantics and Compiler Error Recovery.* PhD thesis, Computer Science Division—EECS, University of California, Berkeley, California, 94720, June 1985. Report No. UCB/CSD 85/251.

[30] Johann de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.

[31] Saumya K. Debray and David S. Warren. Automatic mode inference for Prolog programs. In *1986 Symposium on Logic Programming*, pages 78–88. IEEE, 1986.

[32] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with applications to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages* [2], pages 105–116.

[33] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography.* Number 323 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 1988.

[34] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2:119–155, 1985.

[35] F. DeRemer and R. Jullig. Tree-affix dendrogrammars for languages and compilers. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 300–319, Berlin, Heidelberg, New York, 1980. Springer-Verlag.

[36] F. L. DeRemer. Transformational grammars. In F. L. Bauer and J. Eickel, editors, *Compiler Construction; An Advanced Course*, pages 109–120. Springer-Verlag, Berlin, Heidelberg, New York, second edition, 1976.

[37] Thierry Despeyroux. Executable specification of static semantics. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, number 173 in Lecture Notes in Computer Science, pages 215–233, Berlin, Heidelberg, New York, 1984. Springer-Verlag.

[38] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structure editors: the MENTOR experience. Research Report No. 26, INRIA, July 1980.

[39] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy. A structure oriented program editor: a first step towards computer assisted programming. In E. Gelenbe and D. Poiter, editors, *International Computing Symposium 1975*, pages 113–120. North-Holland Publishing Company, 1975.

[40] J. Doyle. A truth maintenance system. In Webber and Nilsson [137], pages 496–516.

[41] G. Engels, M. Nagl, and W. Schaefer. On the structure of structure-oriented editors for different applications. In Henderson [55], pages 190–198.

[42] Robert E. Filman. Reasoning with worlds and truth maintenance. *Communications of the ACM*, 31(4):382–401, April 1988.

[43] Michael J. Fischer and Richard E. Ladner. Data structures for efficient implementation of sticky pointers in text editors. Technical Report 79-06-08, Department of Computer Science, University of Washington, Seattle, Washington, 98195, June 1979.

[44] E. R. Gansner, J. R. Horgan, D. J. Moore, P. T. Surko, D. E. Swartwout, and J. H. Reppy. SYNED—a language-based editor for an interactive programming environment. In *IEEE Spring Compcon '83*, 1983.

[45] Harald Ganzinger, Robert Giegerich, Ulrich Mönke, and Reinhard Wilhelm. A truly generative semantics-directed compiler generator. In *Proc. of the SIGPLAN '82 Symposium on Compiler Construction*, pages 172–184, Boston, Massachusetts, June 25–27, 1982. ACM. Appeared as Sigplan Notices, 17(6), June 1982.

[46] Harald Ganzinger and Michael Hanus. Modular logic programming of compilers. In *IEEE Symposium on Logic Programming*, 1985.

[47] Phillip Garrison. *Modeling and Implementation of Visibility in Programming Languages.* PhD thesis, Computer Science Division—EECS, University of California, Berkeley, California, 94720, December 1987. Report Number UCB/CSD 88/400.

[48] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the ACM*, 32(2):280–295, April 1985.

[49] Susan L. Graham. On bounded right context languages and grammars. *SIAM Journal on Computing*, 3(3):224–254, 1974.

[50] James N. Gray and Michael A. Harrison. On the covering and reduction problems for context-free grammars. *Journal of the ACM*, 19(4):675–698, October 1972.

[51] Wilfred J. Hansen. *Creation of Hierarchic Text with a Computer Display.* PhD thesis, Stanford University, June 1971.

[52] Wilfred J. Hansen. User engineering principles for interactive systems. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, chapter 8, pages 217–231. McGraw-Hill, 1984.

[53] Robert Harper. Introduction to standard ML. Technical report, Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1988.

[54] Peter Henderson, editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984.

[55] Peter Henderson, editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1986.

[56] Peter Henderson, editor. *ACM SIGSOFT '88: Third Symposium on Software Development Environments*, 1988.

[57] Roger Hoover and Tim Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *Proc. of the SIGPLAN '86 Symposium on Compiler Construction*, pages 39–50, Palo Alto, California, June 25–27, 1986. ACM. Appeared as Sigplan Notices, 21(7), July 1986.

[58] M. R. Horton. *Design of a multi-language editor with static error detection capabilities.* PhD thesis, Computer Science Division—EECS, University of California, Berkeley, California, 94720, 1981.

[59] Susan Horwitz and Tim Teitelbaum. Generating editing environments based on relations and attributes. *ACM Trans. on Programming Languages and Systems*, 8(4):577–608, October 1986.

[60] Paul Hudak and Philip Wadler. Report on the functional programming language Haskell. Draft proposed standard, December 1988.

[61] H. B. Hunt, D.J. Rosenkrantz, and T.G. Szymanski. The covering problem for linear context-free grammars. *Theoretical Computer Science*, 2:361–382, 1976.

[62] Fahimeh Jalili and Jean H. Gallier. Building friendly parsers. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages* [3], pages 196–206.

[63] G. F. Johnson and C. N. Fischer. Non-syntactic attribute flow in language-based editors. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages* [3], pages 185–193.

[64] Guy L. Steele Jr. *The Definition and Implementation of a Computer Programming Language based on Constraints.* PhD thesis, Massachusetts Institute of Technology, 1980.

[65] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, 1984.

[66] G. Kahn. Natural semantics. In F. Brandenburg, G. Vidal-Nacquet, and W. Wirsig, editors, *STACS '87: Fourth Annual Symposium on Theoretical Aspects of Computer Sciences*, Lecture Notes in Computer Science, pages 22–39, Berlin, Heidelberg, New York, 1987. Springer-Verlag.

[67] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3:152–188, 1983.

[68] Gail E. Kaiser. *Semantics for Structure Editing Environments.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, May 1985.

[69] Gail E. Kaiser, Peter H. Feiler, Fahimeh Jalili, and Johann H. Schlichter. A retrospective on DOSE: An interpretive approach to structure editor generation. *Software—Practice & Experience*, 18(8):733–748, 1988.

[70] Gail E. Kaiser and Elaine Kant. Incremental parsing without a parser. *Journal of Systems and Software*, 5(2):121–144, May 1985.

[71] Simon M. Kaplan. Incremental attribute evaluation on graphs. Manuscript, 1986.

[72] Simon Mark Kaplan. *Specification and Verification of Context Conditions for Programming Languages.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.

[73] Uwe Kastens. Personal communication.

[74] Peter Andre Christopher Kirslis. *The SAGA editor: A Language-Oriented Editor Based on an Incremental LR(1) Parser.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1986.

[75] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[76] D. E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[77] Robert A. Kowalski and Kenneth A. Bowen, editors. *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, Cambridge, Massachusetts and London, England, 1988. The MIT Press.

[78] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller Pedersen, and Kristen Nygaard. An algebra for program fragments. In *Proc. of the SIGPLAN '86 Symposium on Language Issues in Programming Environments* [4], pages 161–170. Appeared as Sigplan Notices, 20(7), July 1985.

[79] J. Krohnfeldt and R. Kessler. Microscope—rule-based analysis of programming environments. In *Proceedings of the Second Conference on Artificial Intelligence Applications*, December 1985.

[80] Wilf R. LaLonde. Regular right-part grammars and their parsers. *Communications of the ACM*, 20(10):731–741, October 1977.

[81] Bernard Lang. On the usefulness of syntax directed editors. In Conradi et al. [27], pages 47–51.

[82] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. The Ergo support system: An integrated set of tools for prototyping integrated environments. In Henderson [56], pages 25–34.

[83] William Leler. *Constraint Programming Languages: Their Specification and Implementation.* Addison-Wesley, Reading, Massachusetts, 1988.

[84] Mark A. Linton. Implementing relational views of programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132–140, 1984.

[85] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, New York, second, extended edition, 1987.

[86] L. M. Masinter. Global program analysis in an interactive environment. Technical Report SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, 1980.

[87] D. A. McAllester. An outlook on truth maintenance. AI Memo No. 551, Massachusetts Institute of Technology Artificial Intelligence Laboratory, January 1980.

[88] D. V. McDermott and G. J. Sussman. The Conniver reference manual. AI Memo No. 259, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, May 1972.

[89] Drew McDermott. Contexts and data dependencies: A synthesis. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-5(3):237–246, May 1983.

[90] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Trans. on Software Engineering*, SE-7(5):472–481, 1981.

[91] Raul Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, March 1982.

[92] C. S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 1(1):43–66, 1985.

[93] Paul H. Morris and Robert A. Nado. Representing actions with an assumption-based truth maintenance system. In *Proc AAAI-86 (Fifth National Conference on Artificial Intelligence)*, pages 13–17, 1986.

[94] Lee Naish. *Negation and Control in Prolog.* Number 238 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 1986.

[95] Lisa Rubin Neal. Cognition-sensitive design and user modelling for syntax-directed editors. In *CHI+GI*, pages 99–102, 1987.

[96] Anton Nijholt. *Context-Free Grammars: Covers, Normal Forms, and Parsing*, volume 93 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[97] M. Nilsson. Worlds shortest Prolog interpreter? In J. A. Campbell, editor, *Implementations of Prolog*, pages 87–92. Ellis Horwood Ltd., 1984.

[98] Martin Nilsson. FOOLOG—a small and efficient Prolog interpreter. UPMAIL Technical Report 20, Uppsala Programming Methodology and Artificial Intelligence Laboratory, Department of Computing Science, Uppsala University, Uppsala, Sweden, June 1983.

[99] Robert E. Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3/4):225–236, 1985.

[100] Robert L. Nord and Frank Pfenning. The Ergo attribute system. In Henderson [56], pages 110–120.

[101] David Notkin. *Interactive Structure-Oriented Computing.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, February 1984.

[102] David Notkin. The GANDALF project. *Journal of Systems and Software*, 5(2):91–106, May 1985.

[103] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[104] Gustav Pomberger. *Software Engineering and Modula-2.* Prentice-Hall, Englewood Cliffs, N.J., 1984.

[105] Dag Prawitz. *Natural Deduction: A Proof-Theoretic Study.* Almquist and Wiksell, Stockholm, 1965.

[106] Roger S. Pressman. *Software Engineering: A Practicioner's Approach.* McGraw-Hill, second edition, 1987.

[107] Steven P. Reiss. Generation of compiler symbol processing mechanisms from specifications. *ACM Trans. on Programming Languages and Systems*, 5(2):127–164, April 1983.

[108] Steven P. Reiss. Graphical program development with PECAN program development system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 30–41, 1984.

[109] Steven P. Reiss. GARDEN tools: Support for graphical programming. In Conradi et al. [27], pages 59–72.

[110] Ray Reiter. On closed world data bases. In Webber and Nilsson [137], pages 119–140.

[111] T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages* [3], pages 169–176.

[112] Thomas Reps, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language-based editors. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, St. Petersburg Beach, Florida, January 13–15, 1986. ACM.

[113] Thomas Reps and Tim Teitelbaum. The synthesizer generator. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, 1984.

[114] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Ithaca, New York 14853, August 1985.

[115] Thomas W. Reps. *Generating Language-Based Environments.* MIT Press, 1984.

[116] Charles Rich and Richard C. Waters. The Programmers Apprentice: A research overview. *IEEE Computer*, 21(11):10–25, November 1988.

[117] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:32–41, 1965.

[118] D. J. Rosenkrantz and H. B. Hunt. Efficient algorithms for automatic construction and compactification of parsing grammars. *ACM Trans. on Programming Languages and Systems*, 9(4), 1987.

[119] Alan Rosselet. *Definition and Implementation of Context Conditions for Programming Languages.* PhD thesis, Computer Systems Research Institute, University of Toronto, Toronto, Canada M5S 1A1, 1984.

[120] Johns F. Rulifson. QA4 programming concepts. Artificial Intelligence Group Technical Note 60, Stanford Research Institute, August 1971. SRI Project 8721.

[121] E. Sandewall. Programming in the interactive environment: The LISP experience. *ACM Computing Surveys*, 10(1):32–72, March 1978.

[122] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Trans. on Graphics*, 5(2):79–109, April 1986.

[123] U. Shani. Should program editors not abandon text oriented commands? *SIGPLAN Notices*, 18(1):35–41, 1983.

[124] R. M. Stallman. EMACS manual for TWENEX users. AI Memo 555, Artificial Intelligence Laboratory, Massachussetts Institute of Technology, Cambridge, 1980.

[125] R. M. Stallman. EMACS: the extensible, customizable, self-documenting display editor. In *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* [1], pages 147–156. Appeared as Sigplan Notices, 16(6), June 1981.

[126] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, Cambridge, Massachusetts and London, England, 1986.

[127] Sun Microsystems, Inc. *Windows and Window Based Tools: Beginner's Guide*, 1986.

[128] Gerald J. Sussman and Drew Vincent McDermott. From planner to conniver—a genetic approach. In *Fall Joint Computer Conference 1972*, pages 1171–1179, 1972.

[129] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[130] T. Teitelbaum and T. Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[131] Pascal Van Hentenryk. *Constraint Satisfaction in Logic Programming.* Massachusetts Institute of Technology, Cambridge, Massachusetts, 1989.

[132] Michael L. Van De Vanter. User-centered program viewing. Research Proposal, November 1987.

[133] Scott Vorthmann. Incremental consistency maintenance in a syntax-directed programming environment. Technical Report GIT-ICS-87/46, School of Information and Computer Science, Georgia Institute of Technology, May 1987.

[134] Vance E. Waddle. Production trees: A compact representation of parsed programs. Technical report, IBM Thomas J. Watson Research Center, 1986.

[135] R. C. Waters. Program editors should not abandon text oriented commands. *SIGPLAN Notices*, 17(7):39–46, 1982.

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

[137] Bonnie Lynn Webber and Nils J. Nilsson, editors. *Readings in Artificial Intelligence.* Tioga, Palo Alto, California, 1981.

[138] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[139] Terry Winograd. Beyond programming languages. *Communications of the ACM*, 22(7):391–401, July 1979.

[140] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin, Heidelberg, New York, third, corrected edition, 1985.

[141] A. L. Wolf, L. A. Clarke, and J. C. Wileden. A model of visibility control. *IEEE Trans. on Software Engineering*, 14(4):512–520, April 1988.

[142] S. R. Wood. Z—the 95% program editor. In *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* [1], pages 1–7. Appeared as Sigplan Notices, 16(6), June 1981.

Appendix A

Colander Language Reference

A brief summary of the syntax and primitives of COLANDER is presented. Familiarity with PROLOG and with the material in Chapter 5 is assumed.

A.1 Language Summary

This section summarizes the language COLANDER.

A.1.1 Symbols

Symbols are composed of sequences of alphabetic characters, digits, and the special characters { `+`, `*`, `-`, `=`, `_`, `<`, `>`, `!`, `&`, `@`, `/`, `?` }. Symbols must begin with an alphabetic character. Upper case and lower case alphabetic characters are distinguished.

The symbols that can appear as subtree names in productions are further constrained to be in the set { `a-z`, `A-Z`, `0-9`, `_`, `-` } and must begin with a letter.

A.1.2 Variables

Variables are symbols prefixed by the character '?'. No spaces are allowed between the '?' and the remainder of the symbol. The special name '??' is used to represent anonymous variables.

A.1.3 Numbers

COLANDER supports both (signed) integers and real numbers. Scientific notation is not currently implemented.

A.1.4 Strings

Strings are sequences of characters enclosed in double quotes. Thus "`This is a string`" is a string. To embed a '"' character in a string, quote it with the backslash ('\') character.

A.1.5 Lists

Lists are comma-separated sequences of literal terms demarcated with brackets. For example, `[]` represents the empty list, while `[this, is, a, list, with, [a, sublist]]` is a list of 6 elements, the last of which is itself a list of two elements.

The notation `[ ... | ... ]` denotes the COLANDER equivalent of a LISP dotted-pair. It is usually used to decompose lists as in the **append** operation.

A.1.6 Structures

Structures in COLANDER follow the usage of PROLOG.

A.1.7 Comments

Comments in a COLANDER description are delimited by `/* ... */`. The separator `//` used to separate the portions of the definition of an operator also can contain comments—any text following the `//` up to the end of the line is ignored.

A.2 Colander Primitive Functions

This section lists the currently implemented COLANDER primitives.

A.2.1 Prolog-like Functions

The following primitive functions are similar to their PROLOG counterparts.

=(*x, y*)
assert(*fact*)
assert(*fact, datapool*)
bagof(*vars, form, ?result*)
call(*literal*)
cut()
evlisp(*lisp-expr, result*)
fail()
format(*string, arg1, ...*)
if(*test, truegoals, ?falsegoals*)
is(*result, lisp-expr*)
not(*literal*)
retract(*fact*)
retract(*fact, datapool*)
true()

The next two primitives resemble **bagof** and **not**, but have results that are monitored by the consistency manager. Chapter 5 discusses them in detail.

all-solutions(*vars, form, ?result*)
notever(*literal*)

A.2.2 Strings

Primitive functions for manipulating strings.

atof(*string, value*) — converts numeric string to integer value
atoi(*string, value*) – converts numeric string to floating piont value
string/=(x, y)
string<=(x, y)
string<(x, y)
string=(x, y)
string>=(x, y)
string>(x, y)

A.2.3 Arithmetic

Primitive functions for arithmetic.

Operations

*(x, y, *res*)
+(x, y, *res*)
-(x, y, *res*)
/(x, y, *res*)

Arithmetic Comparison

neq(x, y)
leq(x, y)
lt(x, y)
geq(x, y)
gt(x, y)

A.2.4 List Accessors

head(*list, res*)
tail(*list, res*)
last(*list, res*)

A.2.5 Declarations

Global Declarations

fact(*name([arg1, ldots , argN]), [documentation]*)

entity-property(*name*, [*documentation*])
maintained-property(*name*, [*documentation*])
node-property(*name*, [*documentation*])

Local Declarations

context-name(*name*, [*documentation*])
datapool-name(*name*, [*documentation*])
entity-name(*name*, [*documentation*])

A.2.6 Object Allocation

Primitives for allocating objects and propagating context datapools.

new-context(*name, res, child-list, prop1, ...*)
new-context*(*name, res, prop1, ...*)
new-datapool(*name, res, prop1, ...*)
new-entity(*name, res, prop1, ...*)
new-anon-entity(*res, prop1, ...*)

A.2.7 Mapping Functions

The mapping functions are used to map goals over lists of subtrees. They are discussed in Chapter 5.

for-all-children(*op, kid, kid-list, iterator-forms, vars, res*)
for-each-child(*op, kid, kid-list, iterator-forms, vars, res*)

A.2.8 Structural Properties

Structural subtree properties maintained by COLANDER.

child0(*node, res*)
child1(*node, res*)
child2(*node, res*)
child3(*node, res*)
child4(*node, res*)
child5(*node, res*)
child6(*node, res*)
child7(*node, res*)
child8(*node, res*)
child9(*node, res*)
children(*node, res*)
context(*res*)
get-child(*child-index, res*)
is-leftmost-child(*node*)

is-rightmost-child(*node*)
left-sibling(*node, value*)
right-sibling(*node, value*)
pname(*node, res*)
self-node(*x*)

A.3 COLANDER in LADLE

```
LANGUAGE colander

LEXICAL

whitespace = { \t\n\^L} => IGNORE;
comment = "/*" ~ "*/" => SCREEN;
alt_comment = "#" - "\n"  => SCREEN;

string  = "\"" - "\"" ;

identifier = {a-zA-Z}{a-zA-Z0-9:\_\-\+\*\\@&!\?}*;

/* variables are prefixed by a "?" */
variable = ("?"{a-zA-Z}({_a-zA-Z\-0-9}*))|("??");
sep = "//" - "\n" ;

integer = [{-+}]({0-9})+;

real = [{-+}]({0-9}+)"."({0-9}+);

id_assert = "ASSERT" IN identifier;
id_context_name = "CONTEXT-NAME" IN identifier;
id_dp_name = "DATAPOOL-NAME" IN identifier;
id_entity_name = "ENTITY-NAME" IN identifier;
id_entity_property = "ENTITY-PROPERTY" IN identifier;
id_fact = "FACT" IN identifier;
id_for_all = "FOR-ALL-CHILDREN" IN identifier;
id_for_each = "FOR-EACH-CHILD" IN identifier;
id_if = "IF" IN identifier;
id_incl = "INCLUDE" IN identifier;
id_lang = "LANGUAGE" IN identifier;
id_lisp = "LISP" IN identifier;
id_load = "LOAD" IN identifier;
id_maintained = "MAINTAINED-PROPERTY" IN identifier;
id_node_prop = "NODE-PROPERTY" IN identifier;
id_on = "ON" IN identifier;
id_pure = "PURE" IN identifier;
id_retract = "RETRACT" IN identifier;

/* Abstract syntax */
ABSTRACT

description = lang_def outerforms => description
        ;

lang_def =   "LANGUAGE"  identifier  => language_def
        ;
```

```
outerforms = outer_form *
        ;

outer_form =
        /* definitions */
          operator
        | rule
        | goal
        | on_condition

        /* declarations */
        | fact_decl
        | entity_property_decl
        | maintained_property_decl
        | node_property_decl
        | entity_name_decl
        | datapool_name_decl

        /* Declarations for shadowing */
        | pure_decl
        | mode_decl
        | "INCLUDE" string => merge_file
        | "LOAD" string => load_file
        ;

maintained_property_decl =
          "MAINTAINED-PROPERTY" "(" alpha_id opt_doc ")" "." =>
                                                  maintained_property_decl
        ;

node_property_decl =
          "NODE-PROPERTY" "(" alpha_id opt_doc ")" "." =>
                                                  node_property_decl
        ;

entity_name_decl =
         "ENTITY-NAME" "(" alpha_id opt_doc ")" "." => entity_name_decl
        ;

datapool_name_decl =
        "DATAPOOL-NAME" "(" alpha_id opt_doc ")" "." => datapool_name_decl
        ;

context_name_decl = "CONTEXT-NAME"  "(" alpha_id opt_doc ")" "." => context_name_decl
        ;

entity_property_decl =
        "ENTITY-PROPERTY" "(" alpha_id opt_doc ")" "." => entity_decl
        ;
```

```
fact_decl = "FACT" "(" functor opt_doc ")" "." => fact_decl
        ;

pure_decl = "PURE" "(" functor_id ")" "." => pure_decl
        ;

mode_decl = "BEHAVIOR" "(" functor_id "," mode "," mode ")" "." => mode_decl
        ;

mode = mode_symbol+
        ;

mode_symbol = "+"
        | "-"
        ;


on_condition =
            on_assert_kw rule_head "=>" functor_list "." => on_assert
        |   on_retract_kw rule_head "=>" functor_list "." => on_retract
        ;

on_assert_kw = "ON" "ASSERT" => on_assert_kw;
on_retract_kw = "ON" "RETRACT" => on_retract_kw;

opt_doc = [ "," string ]            => opt_document
        ;

goal = ":-" goal_clause "." => goal
        ;

/* goal_clause is introduced so that we can wrap an extra level
** of parenthesis around the list of functors
*/
goal_clause = goal_functor_list + "|"
        ;

/* goal_functor_list is introduced so that we can wrap an extra level
** of parenthesis around the functors in the individual lists.
*/
goal_functor_list = functor_list => goal_functor_list
        ;

rule =      rule_head ":-"  functor_list "." => full_rule
        | rule_head "."   => bodiless_rule
        ;

rule_head = simple_functor
        | eval_form
```

```
        ;

operator = alpha_id "=" operator_def_seq          => abstract_operator
        | alpha_id "==>" "{" op_defs "}"                  => terminal_operator
        ;

operator_def_seq = operator_rhs_definition + "|"
        ;

operator_rhs_definition
        =  prod_rhs  "=>" "{" "}"       => empty_operator
        | prod_rhs  "=>" alpha_id "{" op_defs "}" => semantic_operator
        ;

op_defs = local_decls_part
        | local_decls_part sep first_pass_goals
        | local_decls_part sep first_pass_goals sep second_pass_goals
        ;

prod_rhs = rhs => prod_rhs
        ;

rhs     =
          grammar_symbol_seq                                  => ANNOTATE
        | grammar_symbol_seq "+" opt_grammar_symbol           => rhs_plus
        | grammar_symbol_seq "++" opt_grammar_symbol          => rhs_plural
        | grammar_symbol_seq "*" opt_grammar_symbol           => rhs_star
        | "[" grammar_symbol_seq "]"                          => rhs_optional
        ;

grammar_symbol_seq = grammar_symbol* => TREE
        ;

grammar_symbol =
          alpha_id
        | string
        ;

opt_grammar_symbol = [ grammar_symbol ]
        ;


local_decls_part = local_decl* => local_decls_part
        ;

local_decl =
          entity_name_decl
        | datapool_name_decl
        | context_name_decl
        | rule
```

```
        ;

first_pass_goals = goal_seq => first_pass_goals
        ;

second_pass_goals =  goal_seq => second_pass_goals
        ;

goal_seq = goal*
        ;


/*
** FUNCTORS
*/
functor_list = functor* ","
        ;

/* This reduction is needed to properly prepass an if-then-else.
** It's a sloppy hack that should be fixed in a better way.... someday.
*/
if_functor_list = functor_list => if_seq
        ;

sequence = "(" functor_list ")"   => sequence
        ;

functor =  rule_head
        |  rule_head ":" message  => msg_functor
        ;

message = string                                  => msg1
        | "(" string ")"                          => msg2
        | "(" string "," message_arg_seq ")"      => msg
        ;

message_arg_seq = literal + ","
        ;

eval_form = "<"  simple_functor "," datapool_literal ">" => eval_functor
        |   "<" variable  ","  datapool_literal ">" => indirect_eval_functor
        ;

simple_functor =
          functor_id "(" literal_list ")"         => op_functor
        | conditional
        | iterator
        | evlisp
        | cut                                     => cut_functor
        ;
```

```
functor_id =  alpha_id
        | symbolic_id                    => symb_id
        ;

symbolic_id   =
          "+"
        | "-"
        | "*"
        | "/"
        | ">="
        | "<"
        | ">"
        | "="
        | "<="
        | "/="
        ;

alpha_id = identifier
        | "ON"
        | "ASSERT"
        | "RETRACT"
        | "LANGUAGE"
        | iter_kw
        ;


/* Literals and special syntax */

literal_list =  literal * ","
        ;

nonempty_literal_list =literal + ","
        ;

literal = simple_functor
        | alpha_id
        | eval_form
        | string
        | real
        | integer
        | variable
        | rail
        | node_ref
        | sequence
        ;


datapool_literal = alpha_id
        | variable
```

```
        ;

rail    = "[" literal_list "]"      => prolog_list
        | "[" literal_list "|" literal_list "]" => prolog_dotted_list
        ;

node_ref = "$" alpha_id  => node_reference
        | "$" alpha_id "<" integer ">"    => indexed_node_reference
        ;

/*
** Conditionals
*/
conditional = "IF"  "(" if_functor_list ";" if_functor_list
                                ";" if_functor_list ")" =>          if_then_else
        ;

/*
** ITERATORS
*/
iterator = iter_name "(" op_or_list_kw  ","  /*operator*/
                         variable "," /*child variable*/
                         vbl_or_rail "," /* list of children */
                         iter_clauses ","
                         vbl_or_rail  "," /* result vars */
                         variable  /* final result */
                       ")"        => iterator

        ;

op_or_list_kw = alpha_id
        | ":list"               => list_id
        ;

iter_name = id_for_all      => for_all
        | id_for_each    => for_each
        ;

vbl_or_rail = variable
        | rail
        ;

iter_clauses = "{" iter_clause_seq "}" => iter_clauses ;

iter_clause_seq = iter_clause + "||"
        ;

/* functor_list here does not allow multiple alternatives IN the top level
** of the iterator goal.
*/
```

```
iter_clause =  iter_kw  "=>" goal_clause => iterator_clause
        ;

iter_kw =  ":all"                => kw_id
        | ":all-but-first"       => kw_id
        | ":all-but-last"        => kw_id
        | ":middle"              => kw_id
        | ":first"               => kw_id
        | ":last"                => kw_id
        | ":rest"                => kw_id
        | ":singleton"           => kw_id
        ;

cut =  "!"              => cut
        ;

/*
** Access to LISP
*/
evlisp   = "LISP" "(" lisp_sexpr "," variable ")" => evlisp
        ;

lisp_sexpr = lisp_literal                  => lisp_lit
        | "(" lisp_seq ")"                 => lisp_list
        | "'" lisp_sexpr                   => lisp_quote
        ;

lisp_seq = lisp_sexpr *
        ;

lisp_literal =
          functor_id
        | string
        | real
        | integer
        | variable
        ;
```

CONCRETE

Appendix B

A Simple Example: Tiny

"*Tiny*" is a simple language having nested block structure, declarations of variables and uses of variables. It is used for teaching purposes and to test the COLANDER implementation. As is usual, it requires definition before use, and it requires that a name not be declared twice in the same scope. Scopes are associated with blocks in the language.

B.1 Tiny in LADLE

```
LANGUAGE tiny

LEXICAL

space = { \t\n\^L} => IGNORE ;

comment = "/*" ~ "*/" => SCREEN ;

id ={a-zA-Z}{a-zA-Z0-9}*;

keyPROG  = 'PROGRAM' IN id;

ABSTRACT

program = 'PROGRAM' block => program
        ;

block = id "{" decls ";"  stmts "}" => block
        ;

decls = declaration + ";"
      ;

stmts = statement + ";"
      ;
```

```
declaration = idlist ":" id   => declaration
            ;

idlist = id + "," => idlist
       ;

statement = assign
          | block
          |
          ;

assign = id ":=" expression                    => assign
       ;

/* Expression is introduced to provide a handle for error recovery */
expression
       = expression "+" expression                    => expr_plus
       | expression "*" expression                    => expr_times
       | "(" expression ")"                           => ANNOTATE
       | id                                           => expr_id
       ;

CONCRETE

expression = top_expr
       ;

top_expr = factor
       | expression "+" factor
       ;

factor  = factor "*" term
       | term
       ;

term    = id
       | "(" expression ")"
       ;
```

B.2 Tiny in COLANDER

```
/*
** This is a very simple example Colander description
**
** The language Tiny has nested block structure, and has 2 kinds of
** interesting statements—definitions and uses.
*/
LANGUAGE tiny

fact(bound(?name, ?entity), "Represents a binding of a name to a variable.").
```

```
fact(blockname(?tname), "Holds the name of the current block.").
fact(parent(?parent), "Parent of current scope.").
fact(scope(?parent), "Current scope in current context.").

fact(contains-block(?tname), "Represents the declaration of a block.").

entity-property(type, "Represents the type of an entity.").

maintained-property(expr-type, "Synthesized type of expression tree").
maintained-property(num-operands, "Number of operands in expression").

/*
** Action to perform when any BOUND fact is removed from the database
*/
on retract <bound(?name,?entity), ?dp> =>
  format("Retracting fact BOUND ~a~%",?name).

/*
** Action to perform when any BOUND fact is added to the database
*/
on assert <bound(?name, ?entity), ?dp> =>
  format("Asserting fact BOUND ~a in ~a~%", ?name, ?dp).

/*
** LOOKUP is a simple search rule for nested block structure.
**
** The symbol "nil" here could be any literal symbol provided that it is
** used consistently.
*/
<lookup(?term), nil> :- !, fail().
<lookup(?term), ?pool> :- <?term, ?pool>.
<lookup(?term), ?pool> :- <parent(?parent), ?pool>,
                          <lookup(?term), ?parent>.

/* TYPE mapping functions
** These could either be procedures or facts; take your pick.
*/
result-type("integer", "integer", "integer").
result-type("integer", "float",   "float").
result-type("float",   "float",   "float").
result-type("float",   "integer", "integer").


program = PROGRAM block => program {
        context-name(root-context).
        datapool-name(root-scope).
        //
        :- new-context*(root-context, ?nc),
           new-datapool(root-scope, ?scope),
           assert(scope(?scope), ?nc),
```

```
            assert(parent(nil), ?scope),
            assert(blockname("main program"), ?scope).

}

assign = id ":=" expression => assign {
        //
        //
        :- context(?context),
            <scope(?scope), ?context>,
            pname($id, ?name),
            <lookup(bound(?name, ?entity)), ?scope>:
                                        ("~a is not bound!", ?name),

            expr-type($expression, ?expr-type),
            type(?entity, ?id-type),
            =(?id-type, ?expr-type) :
                  ("Type-mismatch! id = ~a, expr = ~a~%",
                                    ?id-type, ?expr-type).
}

declaration = idlist ":" id  => declaration {
        //
        //
        :- pname($id, ?tname),
            context(?context),
            <scope(?scope), ?context>,
            children($idlist, ?kids),
            for-all-children(id, ?child, ?kids,
              { :all => pname(?child, ?name),
                        not(<bound(?name, ??), ?scope>):
                          ("~a is already bound in scope ~a! ~%", ?name, ?block),
                        new-entity(binding, ?ent),
                        assert(bound(?name, ?ent), ?scope),
                        assert(type(?ent, ?tname))
             },  ?name, ?namelist).
}

id = => id {
        entity-name(binding).
}

idlist = id + => idlist {
}


/*
** A block has to create a new scope and pass it to its children.
** It also ensures that two blocks with  the same name are detected.
*/
```

```
block = id BEGIN decls stmts END => block {
        context-name(block-context).
        datapool-name(local-scope).
        //
        :- new-context*(block-context, ?newcon),
           context(?context),
           <scope(?parent), ?context>,
           new-datapool(local-scope, ?scope),
           assert(scope(?scope), ?newcon),
           assert(parent(?parent), ?scope),
           pname($id, ?blockname),
           assert(blockname(?blockname), ?scope).

        //
        :- context(?context),
           <scope(?outer), ?context>,
           pname($id, ?name),
           not(<contains-block(?name), ?outer>) :
                        "Illegal to declare a block twice in the same scope! ~%",
           assert(contains-block(?name), ?outer).

        }


expr_plus =  expression "+" expression     => expr_plus {
        num-operands($expr_plus, ?nop) :-
                num-operands($expression<1>, ?n1),
                num-operands($expression<2>, ?n2),
                +(?n1, ?n2, ?temp),
                +(?temp, 1, ?nop).

        expr-type($expr_plus, ?type) :-
                expr-type($expression<1>, ?t1),
                expr-type($expression<2>, ?t2),
                result-type(?t1, ?t2, ?type).
        //
        //
}

expr_times = expression "*" expression     => expr_times {
        num-operands($expr_times, ?nop) :-
                num-operands($expression<1>, ?n1),
                num-operands($expression<2>, ?n2),
                +(?n1, ?n2, ?temp),
                +(?temp, 1, ?nop).

        expr-type($expr_times, ?type) :-
                expr-type($expression<1>, ?t1),
                expr-type($expression<2>, ?t2),
                result-type(?t1, ?t2, ?type).
```

```
        //
        //
}

expr_id  = id => expr_id {
        num-operands($expr_id, 1).

        expr-type($expr_id, ?type) :-
                pname($id, ?name),
                context(?context),
                < scope(?scope), ?context >,
                <lookup(bound(?name, ?entity)), ?scope>:
                        ("~a is not declared! ~%", ?name),
                type(?entity, ?type).

        //
        //
        :- pname($id, ?name),
           context(?context),
           < scope(?scope), ?context >,
           <lookup(bound(?name, ?entity)), ?scope>:
              ("~a is an undeclared identifier!", ?name).
}
```

Appendix C

Modula-2 in Colander

Root File of Description

LANGUAGE modula2

LOAD "modula2-lisp"

INCLUDE "axioms"
INCLUDE "vis-axioms"
INCLUDE "type-axioms"
INCLUDE "axioms3"
INCLUDE "modules"
INCLUDE "ie"
INCLUDE "procs"
INCLUDE "types"
INCLUDE "case"
INCLUDE "sets"
INCLUDE "decls"
INCLUDE "expr"
INCLUDE "stmts"

Axioms of Description

/*
Every contour has a type specified by contour-type. The type is used in both scoping rules and for passing the outermost scope of a procedure, function, or module down into nested scopes.

Valid types are:
procedure(?con)
function(?con)
module(?entity, type)

A module's contour has a type "module(entity, type)" that marks it as a module contour, and points back to the entity that owns it.
Type is one of

```
        program
        implementation
        definition
        local
*/
fact(contour-type(?type)).
fact(parent-contour(?env), "Pointer to parent context.").
fact(contour(?contour), "Fact in context that accesses the current contour.").

/* In addition, record contours contain the fact
**      record-contour().
*/
fact(record-contour(), "Present in datapools that contain record field declarations").

/*
** Note—we could make "open-contour" a procedure without any loss of
** generality
*/
fact(open-contour(), "Placed in procedure, with, and for-loop contours.").

/*
A module binding is represented using the "bound" fact that pairs the module
name with an entity.

Every module is represented by an entity. Every entity representing a module
has a mod-contour property. If it is an implementation module, it may also
have a def-contour property associated with it.

*/
entity-property(defn-contour, "The definition contour of a module.").
entity-property(impl-contour, "The implementation contour of a module.").

entity-property(type-of,"Denotes the type of an entity.").
entity-property(mode-of,"Contains the mode of an entity.").
entity-property(formal-type, "Contains var/val if formal parameter type").
entity-property(owner).

maintained-property(expr-type, "The expression type of a node in the tree.").
maintained-property(expr-mode, "The expression mode of a node in the tree.").

fact(bound(name, entity, text-loc), "Represents a binding.").
fact(imported-name(name, entity, text-loc), "Represents the importation of a name.").
fact(export-unqualified(name,entity, text-loc), "Represents the unqualified export of a name.").
fact(export-qualified(name,entity, text-loc), "Represents the unqualified export of a name.").

fact(case-label-type(type-entity),
        "Used in CASE and RECORD structures to encode the type for the labels").

maintained-property(qual-id-entity, "Entity representing a qualified id.").
maintained-property(selector-type,  "Type of selector field in variant record.").
```

```
node-property(lpool).
fact(designator-head(name)).
maintained-property(designator-entity).
fact( unop-compatible(type, operation),          "Table of unary operation coercions.").

/* Following are structures for representing constant  values
        ptr-constant(value)  - Structure name for for pointer constants
        enum-constant(enum-const-entity, enum-type-entity)
                        - Structure name for enumeration constants
        int-constant(value)  - Structure name for "INTEGER" constants
        real-constant(value) - Structure name for real constants
        bool-constant(value) - Structure name for boolean constants
        string-constant (value) - Structure name for string constants
        set-constant(values)          - list-of-values
*/

/*
** The rule constant checks that a mode is constant.
** This is really a rule concerning the knowledge representation
** scheme. The second argument is the value of the constant.
*/
extract-value(ptr-constant(?val), ?val).
extract-value(int-constant(?val), ?val).
extract-value(bool-constant(?val), ?val).
extract-value(string-constant(?val), ?val).
extract-value(real-constant(?val), ?val).
extract-value(set-constaint(?val), ?val).

pure(extract-value).

constant-value(enum-constant(?e, ?type-e), ?val) :-
        !,
        type-of(?type-e, enumeration-type(?list)),
        lisp((position ?e ?list), ?val).

constant-value(?x, ?val) :-
   extract-value(?x, ?val).

/*
* Type descriptors:
*
        pointer-type(type-entity)
        record-type (datapool)
        subrange-type (base-type-entity, low, high)

        procedure-type(list-of-arguments)
        function-type(list-of-arguments, result)
        named-type(type-entity)
        set-type (type-entity,low,high)
```

```
        array-type(component-type, index-type).
        module-type(datapool)
        enumeration-type(list-of-ids)
        opaque-type(id)
*/

type-descriptor(?entity, ?type) :- mode-of(?entity, type), !, type-of(?entity, ?type).
type-descriptor(?entity, ?type) :- type-of(?entity, ?type-e), type-of(?type-e, ?type).

/* These maintained properties are used for procedure declarations */
maintained-property(proc-name).
maintained-property(proc-formals).
maintained-property(func-type).

/*
** These next assertions are performed only once to initialize the global
** datapool.
*/

/*
** type "INTEGER". The type of the "INTEGER" entity is a subrange of itself.
**
*/
:- new-anon-entity(?e, ( mode-of(?e, type),
                         type-of(?e, INTEGER),
                         owner(?e, PREDEFINED))),
    assert(bound( "INTEGER", ?e, PREDEFINED)).

/*
** type CARDINAL
*/
:- new-anon-entity(?e,  ( mode-of(?e, type),
                          type-of(?e, CARDINAL),
                          owner(?e, PREDEFINED)) ),
   assert(bound( "CARDINAL", ?e, PREDEFINED)).

/*
** type REAL
*/
:- new-anon-entity(?e,
                   (mode-of(?e, type),
                    type-of(?e, REAL),
                    owner(?e, PREDEFINED))),
      assert(bound( "REAL", ?e, PREDEFINED)).

/*
*/
:- new-anon-entity(?e,
                       (mode-of(?e, type),
                        type-of(?e, CHAR),
```

```
                                owner(?e, PREDEFINED))),
    assert(bound("CHAR", ?e, PREDEFINED)).

/*
** Type BITSET
*/
:-
    new-anon-entity(?e,
                    ( mode-of(?e, type),
                      type-of(?e, set-type(BITSET,  0, 31)),
                      owner(?e, PREDEFINED))),
    assert(bound("BITSET", ?e, PREDEFINED)).

/*
** Booleans
**
*/
:-
    new-anon-entity(?type-e,
                     ( mode-of(?type-e, type),
                       type-of(?type-e, BOOLEAN),
                       owner(?type-e, PREDEFINED))),

    assert(bound( "BOOLEAN", ?type-e, PREDEFINED)),

    new-anon-entity(?true-e,
                   (mode-of(?true-e, enum-constant(?true-e, ?type-e)),
                    type-of(?true-e, ?type-e)
                    )),
    assert(bound( "TRUE", ?true-e, PREDEFINED)),

    new-anon-entity(?false-e,
                   (mode-of(?false-e, enum-constant(?false-e, ?type-e)),
                    type-of(?false-e, ?type-e))),
    assert(bound("FALSE", ?false-e, PREDEFINED)).
```

Visibility Rules

```
/*
** A name is imported if
**       1.) It is explicitly imported
**       2.) It is declared in a definition contour that has been imported.
**       3.) It is exported (unqualified) from a local contour that has
**    been imported.
*/
<name-is-imported(?name, ?entity, ?text-loc), ?contour> :-
        <imported-name(?name, ?entity, ?text-loc), ?contour>.

/* Works for local modules as well as external ones! */
```

```
<name-is-imported(?name, ?entity, ?text-loc), ?contour> :-
        <imported-name(?mod-name, ?mod-entity, ??), ?contour>,
        mode-of(?mod-entity, module),
        /* Program and local modules do not have a defn-contour */
        defn-contour(?mod-entity, ?mod-con),
        !,
        <bound(?name, ?entity, ?text-loc), ?mod-con>.

/* This clause handles imported local modules */
<name-is-imported(?name, ?entity, ?text-loc), ?contour> :-
        <imported-name(?mod-name, ?mod-entity, ??), ?contour>,
        mode-of(?mod-entity, module),
        impl-contour(?mod-entity, ?mod-con),
        <contour-type(module(??, local)), ?mod-con>,
        <export-unqualified(?name, ?entity, ?text-loc), ?mod-con>.

/* Resolve-binding is used to look for a binding.
*/
<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        <bound(?name, ?entity, ?text-loc), ?contour>.

<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        <open-contour(), ?contour>,
        !,
        <parent-contour(?parent), ?contour>,
        <resolve-binding(?name, ?entity, ?text-loc), ?parent>.

<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        <contour-type(module(?mod-e, ??)), ?contour>,
        <name-is-imported(?name, ?entity, ?text-loc), ?contour>.

<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        <contour-type(module(?mod-entity, implementation)), ?contour>,
        defn-contour(?mod-entity, ?def-con),
        <resolve-binding(?name, ?entity, ?text-loc), ?def-con>.

/* Last, check the standard names */
<resolve-binding(?name, ?entity, ?text-loc), ?contour> :-
        global-dp(?dp),
        <bound(?name, ?entity, ?text-loc),  ?dp>.

<resolve-primitive(?name, ?entity, ?text-loc), ?contour> :-
        <resolve-binding(?name, ?entity, ?text-loc), ?contour>.

/* Make resolve-primitive a pure procedure to eliminate combinatorial
** explosion in the shadowing analysis.
** This is safe since resolve-binding is used in more general contexts
*/
pure(resolve-primitive).
```

```
/*
** DECLARABLE(?name)
*/

/*
** A name is declarable in a contour if
**        1. it is not already bound there.
**        2. the contour is a module contour, and it is not imported there.
**        3. The contour is an implemenation module contour, and the name is
** not imported into the corresonding definition module.
*/

<declarable(?name), ?contour> :-
    not(<bound(?name, ?entity, ?text-loc), ?contour>).

<declarable(?name), ?contour> :-
    <contour-type(module(??, ??)), ?contour>,
    not(<imported-name(?name, ?entity, ?text-loc), ?contour>).

<declarable(?name), ?contour> :-
    <contour-type(module(?mod-ent, implementation)), ?contour>,
    defn-contour(?mod-ent, ?def-con),
    not(<imported-name(?name, ?entity, ?text-loc), ?def-con>).
```

Type Axioms

```
/*
** A predicate on type representations. It recurses down the chain of named
** types to reach either a predefined type or a non-primitive defined type.
*/
base-type(named-type(?e), ?bt) :-    base-type(?e, ?bt).
base-type(?e,?e).

/*
** A subrange type must be defined as
**        a subrange of another subrange type
**        a subrange of an enumeration type
**        a subrange of a basic type except(real).  Since the basic types
**         (except real) are internally represented as subranges, this
**         check is pretty straightforward.
**
*/
valid-subrange-type(subrange-type(?type, ??, ??)).
valid-subrange-type(enumeration-type(?type)).
valid-subrange-type(named-type(?type)) :- valid-subrange-type(?type).

valid-set-type(?t) :-
        type-descriptor(?t, subrange-type(?type, ?low, ?high)),
        <=(0, ?low),
```

```
        <=(?high, 31).

valid-set-type(?t) :-
        type-descriptor(?t, enumeration-type(?type)),
        lisp((list-length ?type), ?len),
        <=(?len, 31).

valid-set-type(?t) :-
        type-of(?t, named-type(?type)),
        valid-subrange-type(?type).

/*
** An array index type must be
**      (1) an enumeration
**      (2) a subrange
**      (3) one of the basic types BOOLEAN or CHAR.
**
** Note that the data representation used in this example
** subsumes BOOLEAN into (1) and CHAR into (2)
*/
valid-index-type(enumeration-type(??)).
valid-index-type(subrange-type(??, ??, ??)).
valid-index-type(named-type(?type)) :- valid-index-type(?type).

/*
** valid-index-type is pure because it doesn't access any data values.
*/
pure(valid-index-type).

/*
** Type compatibility — definition of same-type is not quite the same
** as in Wirth's report
*/
same-type(?type, ?type).
same-type(?t1, ?t2) :- type-descriptor(?t1, named-type(?t2)).

/*
** EXPRESSION COMPATIBLE(?t1, ?mode1, ?type2, ?mode2, ?restype)
**
** Two expressions having types t1 and t2 are called expression compatible if
** 1.  t1 and t2 are the same type.
** 2. t1 is a subrange of t2 or t2 is a subrange of t1 or both are subranges
** of the same basis data type.
** 3. One expression has the type "INTEGER" or CARDINAL and the other
** expression is a constant in the interval [0..MAXINT].
** 4. One of the expressions is the constant NIL and the other is an
** arbitrary pointer type.
** 5. t1 and t2 are both procedure types and either both are functions with
** the same function type, or else are not functions, they have the
** same number of parameters, the corresponding parameters are both
```

```
** VAR parameters and have the same type or are VALUE parameters
** and are assignment compatible types.
**
** 6. Both expressions are strings of the same length.
**
** Note that both type and mode information is needed since the mode is
** what tells whether an expression is constant.
**
** Expressions always have both a type and a mode.
** The general form is
** expression-compatible(?t1, ?mode1, ?t2, ?mode2, ?result-type).
** The clauses are reordered from the above definition, and much of the
** equality testing is moved into the choice of heads.
**
** Integer constants are always type-compatible with INTEGER.
**
*/

/*
** 1. ?type1  = ?type2,
** Types are identical.
*/
expression-compatible(?type1, ?mode1, ?type2, ?mode2, ?type1) :-
    same-type(?type1, ?type2).

/*
** 2. Subranges.
*/
expression-compatible(?type1, ?mode1, ?subtype, ?mode2, ?type1) :-
        type-descriptor(?subtype, subrange-type(?type2, ??, ??)), !,
    same-type(?type1, ?type2).

expression-compatible(?subtype, ?mode1, ?type2, ?mode2, ?type1) :-
    type-descriptor(?subtype, subrange-type(?type1, ??, ??)), !,
    same-type(?type1, ?type2).

/*
** Cardinal constants. Integer constants are always of type INTEGER and
** so will be handled correctly above.
*/
expression-compatible(?type1, int-constant(?val), ?type2, ?mode2, ?type2) :-
    type-descriptor(?type2, CARDINAL), !,
    <=(0, ?val).

expression-compatible( ?type1, ?mode1, ?type2, int-constant(?val), ?type1) :-
    type-descriptor(?type1, CARDINAL), !,
    <=(0, ?val).

/*
** 3. Pointers
```

```
*/
expression-compatible(?type1, ?mode1, ?type2, ptr-constant(NIL), ?type1) :-
        type-descriptor(?type1, pointer-type(??)),
        type-descriptor(?type2, pointer-type(??)).

expression-compatible(?type1, ptr-constant(NIL), ?type2, ?mode1, ?type1) :-
        type-descriptor(?type1, pointer-type(??)),
        type-descriptor(?type2, pointer-type(??)).

/*
** 4. Functions
*/
expression-compatible(?type1, ?mode1, type2, ?mode2, ?retype1) :-
        type-descriptor(?type1, function-type(?parms1, ?retype1)), !,
        type-descriptor(?type2, function-type(?parms2, ?retype2)),
        same-type(?retype1, ?retype2),
        formal-types-match(?parms1, ?parms2).
/*
** 5. Procedures
*/
expression-compatible(?type1, ?mode1, type2, ?mode2, ?type1) :-
        type-descriptor(?type1, procedure-type(?parms1)), !,
        type-descriptor(?type2, procedure-type(?parms2)),
        formal-types-match(?parms1, ?parms2).

/* STRINGS
**
*/
expression-compatible(?type1, ?mode1, type2, ?mode2, ?retype1) :-
        type-descriptor(?type1, ?t1),
        is-string(?t1, ?l1),
        type-descriptor(?type2, ?t2),
        is-string(?t1, ?l1).          /* Strings have to be the same length! */

is-string(array-type(?base-e, [ subrange-type(??, 0, ?length) ]), ?length ) :-
        type-descriptor(?base-e, CHAR).

/*
** ASSIGNMENT COMPATIBLE(?contour, ?type1, ?mode1, ?type2, ?mode2),
*/

/*
** The implied order of operation is that a value of type 2 is
** being assigned to an object of type 1.
** An object of data type type1 and an expression of type type2 are assignment
** compatible if:
** type1 and type2 are expression compatible
** One of the two types is "INTEGER" or a subrange thereof and the other
** is CARDINAL or a subrange thereof.
** type1 is of type ARRAY[0..n1] of CHAR, type2 is a constant of type
```

```
** ARRAY[0..n2] of CHAR and n1 >= n2.
**
** The current contour is passed in, because  "INTEGER" and CARDINAL could have
** been redefined...
**
*/

is-type-or-subrange-thereof(?prim-type, ?other-type) :-
        type-descriptor(?other-type, subrange-type(?sub-type, ??, ??)),
        same-type(?prim-type, ?sub-type).

is-type-or-subrange-thereof(?prim-type, ?other-type) :-
        type-descriptor(?other-type, ?p-type),
        same-type(?prim-type, ?p-type).

is-type-or-subrange-thereof(?prim-type, ?other-type) :-
        same-type(?prim-type, ?other-type).

check-assignment-of-primitives(?int-type, ?card-type, ?type1, ?type2) :-
    is-type-or-subrange-thereof(?int-type, ?type1),
    is-type-or-subrange-thereof(?card-type, ?type2).

check-assignment-of-primitives(?int-type, ?card-type, ?type1, ?type2) :-
    is-type-or-subrange-thereof(?card-type, ?type1),
    is-type-or-subrange-thereof(?int-type, ?type2).

pure(check-assignment-of-primitives).

assignment-compatible(?contour, ?type1, ?mode1, ?type2, ?mode2) :-
    expression-compatible(?type1, ?mode1, ?type2, ?mode2, ??).

assignment-compatible(?contour, ?type1, ?mode1, ?type2, ?mode2) :-
    <resolve-primitive("INTEGER", ?int-type, ??), ?contour>,
    <resolve-primitive("CARDINAL", ?card-type, ??), ?contour>,
    check-assignment-of-primitives(?int-type, ?card-type, ?type1, ?type2).

assignment-compatible(?contour, ?type1, ?mode1, ?type2, ?mode2) :-
        is-string(?type1, ?len1),
        is-string(?type2, ?len2),
        <=(?len2, ?len1).

/*
** Formals-match.
*/

valid-actual(variable).
valid-actual(expression).
valid-actual(?mode) :-        constant-value(?mode,??).

pure(valid-actual).
```

```
assignable-loc(variable).
assignable-loc(var-parm).
assignable-loc(val-parm).
pure(assignable-loc).

decomp([?X | [?Y]], ?X, ?Y).

pairlist([], [], []).
pairlist([?h1 | ?r1], [?h2 | ?r2], [pair(?h1, ?h2) | ?Z]) :-
        pairlist(?r1, ?r2,  ?Z).

pure(pairlist).

actuals-match(?contour, ?actuals, ?formals) :-
        pairlist(?actuals, ?formals, ?pairs) :
                "Formals and actuals do not have same length",
        for-all-children(:list, ?pair, ?pairs,
                {:all => actual-matches-formal(?contour, ?pair)}, ??, ??).


pure(actuals-match).

actual-matches-formal(?contour, pair([?act-type, ?act-mode | ??], ?formal)) :-
        type-descriptor(?formal, open-array(?formal-type)),
        !,
        mode-of(?formal, ?formal-mode),
        assignment-compatible(?contour,?formal-type, ?formal-mode, ?act-type, ?act-mode).

actual-matches-formal(?contour,  pair([?act-type, ?act-mode | ??], ?formal)) :-
        mode-of(?formal, var-parm),
        !,
        assignable-loc(?act-mode) :
                "Illegal to pass non-variable as a VAR parameter!",
        type-of(?formal, ?formal-type),
        mode-of(?formal, ?formal-mode),
        assignment-compatible(?contour,?formal-type, ?formal-mode, ?act-type, ?act-mode).

actual-matches-formal(?contour,  pair([?act-type, ?act-mode | ??], ?formal)) :-
        valid-actual(?act-mode) :
                "Argument must be a constant, variable,  or expression",
        type-of(?formal, ?formal-type),
        mode-of(?formal, ?formal-mode),
        assignment-compatible(?contour,?formal-type, ?formal-mode, ?act-type, ?act-mode).

/*
** ?list-of-exprs is a list of pairs [ type mode ] for actual values.
**
*/
deref-array(?contour, ?array-obj, ?list-of-exprs, ?new-type) :-
```

```
        type-descriptor(?array-obj, ?atd),
        check-array-deref(?atd, ?list-of-exprs, ?new-type).

/*
** It is ok to call deref-array pure, since it doesn't seem
** to expose any possible shadowing.
*/
pure(deref-array).

/*
** A "mode" for the formal index value to match the header must be passed to
** assignment-compatible. Take it to be val-parm.
*/
check-array-deref(array-type(?comp, ?index-type), [ [ ?i-type, ?i-mode | ?? ] | ?rest ], ?res) :-
        assignment-compatible(?contour, ?i-type, ?i-mode,  ?index-type, val-parm),
        check-array-deref(?comp, ?rest, ?res).

check-array-deref(?base-type, [], ?base-type).

/* returns either const-true, const-false, or "expression"
** fix this when we do set-types.
*/

formal-types-match(?ftypes, ?actuals) :-
        pairlist(?ftypes, ?actuals, ?pairs) :
                "Definition does not match supplied formal parameters",
        for-all-children(:list, ?pair, ?pairs,
                {:all =>
                        decomp(?pair, ?e1, ?e2),
                        type-of(?e1, ?t1),
                        type-of(?e2, ?t2),
                        same-type(?t1, ?t2) :
                                "Type mismatch in formals-list!",
                        formal-type(?e1, ?m1),
                        mode-of(?e2, ?m1):
                                "Parameter/formal mismatch" },
                ??, ??).

pure(formal-types-match).
/*
** Expressions
**
*/
fact(relation(?name), "table of relational operators").
fact(logic-op(?name), "table of logic operators").
fact(integer-op(?name), "table of basic integer arithmetic operators").
fact(real-op(?name), "table of basic real arithmetic operators").
fact(set-op(?name), "table of basic set operators").
fact(set-rel-op(?name), "table of set relational operators").
```

```
binop-base-type(subrange-type(?type-d, ?low, ?hi), ?res) :-
        binop-base-type(?type-d, ?res).
binop-base-type(named-type(?type-d), ?res) :-
        binop-base-type(?type-d, ?res).
binop-base-type(?res, ?res).

pure(binop-base-type).

binop-compatible(BOOLEAN, ?op) :- logic-op(?op).
binop-compatible(BOOLEAN, ?op) :- !, relation(?op).
binop-compatible(INTEGER, ?op) :- integer-op(?op).
binop-compatible(INTEGER, ?op) :- !, relation(?op).
binop-compatible(CARDINAL, ?op) :- integer-op(?op).
binop-compatible(CARDINAL, ?op) :- !, relation(?op).
binop-compatible(REAL, ?op) :- real-op(?op).
binop-compatible(REAL, ?op) :- !, relation(?op).

binop-compatible( set-type(??, ??, ??), ?op) :- set-op(?op).
binop-compatible( enumeration-type(??), ?op) :- relation(?op).
binop-compatible( subrange-type(??, ??, ??), ?op) :- relation(?op).
binop-compatible( pointer-type(??), "=").
binop-compatible( pointer-type(??), "#").
binop-compatible( pointer-type(??), "<>").
pure(binop-compatible).

:- assert(logic-op("AND")),
   assert(logic-op("OR")),
   assert(logic-op("NOT")),

   assert(integer-op("+")),
   assert(integer-op("-")),
   assert(integer-op("*")),
   assert(integer-op("DIV")),
   assert(integer-op("MOD")),

   assert(real-op("+")),
   assert(real-op("-")),
   assert(real-op("*")),
   assert(real-op("/")),
   assert(real-op("MOD")),

   assert(set-op("+")),
   assert(set-op("-")),
   assert(set-op("*")),
   assert(set-op("/")),

   assert(set-rel-op("=")),
   assert(set-rel-op("#")),
   assert(set-rel-op("<>")),
   assert(set-rel-op("<=")),
```

```
    assert(set-rel-op(">=")),

    assert(relation("=")),
    assert(relation("#")),
    assert(relation("<>")),
    assert(relation("<")),
    assert(relation("<=")),
    assert(relation(">")),
    assert(relation(">=")).

:- assert(unop-compatible( INTEGER,  "-")),
    assert(unop-compatible( INTEGER,  "+")),
    assert(unop-compatible( REAL,  "+")),
    assert(unop-compatible( REAL,  "-")),
    assert(unop-compatible( CARDINAL,  "+")).


/*
** Eval-unop returns a MODE. Note that constant values are modes!
*/
eval-unop("NOT",  bool-constant( TRUE), bool-constant( FALSE)).
eval-unop("NOT",  bool-constant( FALSE), bool-constant( TRUE)).
eval-unop("NOT", ??, expression).
pure(eval-unop).

eval-binop(?op, int-const(?v1), int-const(?v2), int-const(?res)) :-
                    integer-op(?op),
                    lisp((eval-arith-op ?op ?v1 ?v2), ?res).

eval-binop(?op, int-const(?v1), int-const(?v2), bool-const(?res)) :-
                    relation(?op),
                    lisp((eval-bool-op ?op ?v1 ?v2), ?res).

eval-binop(?op, real-const(?v1), real-const(?v2), real-const(?res)) :-
                    real-op(?op),
                    lisp((eval-arith-op ?op ?v1 ?v2), ?res).

eval-binop(?op, real-const(?v1), real-const(?v2), bool-const(?res)) :-
                    relation(?op),
                    lisp((eval-bool-op ?op ?v1 ?v2), ?res).

eval-binop(?op, bool-const(?v1), bool-const(?v2), bool-const(?res)) :-
                    logic-op(?op),
                    lisp((eval-bool-op ?op ?v1 ?v2), ?res).

eval-binop(?op, bool-const(?v1), bool-const(?v2), bool-const(?res)) :-
                    relation(?op),
                    lisp((eval-bool-op ?op ?v1 ?v2), ?res).

eval-binop(?op, set-const(?v1), set-const(?v2), set-const(?res)) :-
```

```
                set-op(?op),
                lisp((eval-set-op ?op ?v1 ?v2), ?res).

eval-binop(?op, set-const(?v1), set-const(?v2), bool-const(?res)) :-
                set-rel-op(?op),
                lisp((eval-set-bool-op ?op ?v1 ?v2), ?res).

pure(eval-binop).

maintained-property(local-binding,
        "Binding between name and entity on id nodes").
id ==> {
        /* entity-name is declared for so that some id nodes can have
        ** entities associated with them. Actual entities are
        ** allocated from mapped goals.
        */
        entity-name(id-entity).

        /* the use of local-binding to cache the binding of the identifier
        ** is a big win here—both for shadowing analysis and during runtime.
        */
        local-binding($id, ?entity) :-
                context(?env),
                <contour(?contour), ?env>,
                pname($id, ?name),
                <resolve-binding(?name, ?entity, ??), ?contour>.
}
```

Module Declarations

```
compilation_unit
        =       definition_module => { }
        |       program_module => { }
        |       impl_module => { }

definition_module
        =       "DEFINITION" "MODULE" id ";"
                    import_seq opt_export definition_seq "END" id "." => def_module
        {
                entity-name(module-ent).
                context-name(mod-context).
                datapool-name(new-scope).

        /* this procedure is local due to embedded allocator */
        <get-module-entity(?name, ?entity), ?contour> :-
                <bound(?name, ?entity, ??), ?contour>.

        <get-module-entity(?name, ?entity), ?contour> :-
                new-entity(module-ent, ?entity,
                                ( mode-of(?entity, module) )).
```

```
        //
        /* Create new entity, scope, and contour and intitialize them. */
        :- global-cm-dp(?context),
            pname($id<1>, ?name),
            <get-module-entity( ?name, ?mod-ent), ?context>,
            new-datapool(new-scope, ?new-scope,
                                    ( parent-contour(?new-scope, ?context),
                                      contour-type(module(?mod-ent, definition)))),
            new-context*(mod-context, ?new-con,
                         ( contour(?new-scope) )).

        //
        :- pname($id<1>, ?name),
            pname($id<2>, ?name) :
                 "Names at beginning and end of module do not agree!~%".
        /*
        ** Establish the bindings
        */
        :- global-cm-dp(?context),
            pname($id<1>, ?name),
            <declarable(?name), ?context>,
            new-scope(?new-scope),
            /* the Module-contour in ?new-scope contains the correct entity */
            <contour-type(module(?mod-ent, ??)), ?new-scope>,
            assert(bound(?name, ?mod-ent, $definition_module), ?context),
            assert(defn-contour(?mod-ent, ?new-scope))
            |
            /* name already declared, but need to associate the
            ** contour with it.
            */
            global-cm-dp(?context),
            pname($id<1>, ?name),
            <bound(?name, ?mod-ent, ?text-loc), ?context>,
            not(defn-contour(?mod-ent, ??)):
                 ("Illegal to define DEFINITION module ~a twice", ?name),
            new-scope(?new-scope),
            assert(defn-contour(?mod-ent, ?new-scope)).

}


impl_module
        =       "IMPLEMENTATION" "MODULE" id
              opt_priority ";" import_seq block id "." => impl_module
        {
            entity-name(module-ent).
            context-name(mod-context).
            datapool-name(new-scope).
```

```
        <get-module-entity(?name, ?entity), ?contour> :-
                <bound(?name, ?entity, ??), ?contour>.

        <get-module-entity(?name, ?entity), ?contour> :-
                new-entity(module-ent, ?entity,
                                ( mode-of(?entity, module) )).

        //
        /* Create new entity, scope, and contour and intialize them. */
        :- global-cm-dp(?context),
           pname($id<1>, ?name),
           <get-module-entity( ?name, ?mod-ent), ?context>,
           new-datapool(new-scope, ?new-scope,
                                ( parent-contour(?new-scope, ?context),
                                contour-type(module(?mod-ent, implementation)))),
           new-context*(mod-context, ?new-con,
                        (contour(?new-scope) )).

        //
        :- global-cm-dp(?context),
           pname($id<1>, ?name),
           <declarable(?name), ?context>,
           new-scope(?new-scope),
           /* the Module-contour in ?new-scope contains the right entity */
           <contour-type(module(?mod-ent, ??)), ?new-scope>,
           assert(bound(?name, ?mod-ent, $impl_module), ?context),
           assert(impl-contour(?mod-ent, ?new-scope))
           |
           /* already declared */
           global-cm-dp(?context),
           pname($id<1>, ?name),
           <bound(?name, ?mod-ent, ?text-loc), ?context>,
           impl-contour(?mod-ent, ?icon) :
                ("Illegal to define IMPLEMENTATION module ~a twice", ?name),
           new-scope(?new-scope),
           assert(impl-contour(?mod-ent, ?new-scope)).

        :- pname($id<1>, ?name),
           pname($id<2>, ?name) :
                "Names at beginning and end of module do not agree!~%".
}

program_module
        =       "MODULE" id opt_priority ";" import_seq block id "." => prog_module
        {
                entity-name(module-ent).
                context-name(mod-context).
                datapool-name(new-scope).

        //
```

```
        /* Create new entity, scope, and contour and intitialize them. */
        :- global-cm-dp(?context),
            new-entity(module-ent, ?mod-ent, ( mode-of(?mod-ent, module))),
            new-datapool(new-scope, ?new-scope),
             assert(parent-contour(?context), ?new-scope),
            assert(contour-type(module(?mod-ent, program)), ?new-scope),
            new-context*(mod-context, ?new-con),
            assert(contour(?new-scope), ?new-con).

        //
        :- global-cm-dp(?context),
            pname($id<1>, ?name),
            <declarable(?name), ?context>:
                                ("Module name ~a is already used!", ?name),
            new-scope(?new-scope),
            module-ent(?mod-ent),
            assert(bound(?name, ?mod-ent, $program_module), ?context),
            assert(impl-contour(?mod-ent, ?new-scope)).

        :- pname($id<1>, ?name),
            pname($id<2>, ?name) :
                  "Names at beginning and end of module do not agree!~%".
}

module_declaration
        =       "MODULE" id opt_priority ";" import_seq opt_export block id ";"
                => module_declaration
        {

                entity-name(module-ent).
                context-name(mod-context).
                datapool-name(new-scope).

        //
        /* Create new entity, scope, and contour and intitialize them. */
        :- context(?context),
            <contour(?contour), ?context>,
            new-entity(module-ent, ?mod-ent, ( mode-of(?mod-ent, module))),
            new-datapool(new-scope, ?new-scope,
                                    ( parent-contour(?contour),
                                      contour-type(module(?mod-ent, local)))),
            new-context*(mod-context, ?new-con,
                         (contour(?new-scope) )).

        //
        :- context(?context),
            <contour(?contour), ?context>,
            pname($id<1>, ?name),
            <declarable(?name), ?contour> :
             ("Module name ~a is already used!", ?name),
```

```
            new-scope(?new-scope),
            module-ent(?mod-ent),
            assert(bound(?name, ?mod-ent, $module_declaration), ?contour),
            assert(impl-contour(?mod-ent, ?new-scope)).

        :- pname($id<1>, ?name),
            pname($id<2>, ?name) :
                "Names at beginning and end of module do not agree!-%".
}

opt_priority
        =       [ "[" const_expr "]" ] => priority { }
```

Import and Export

```
fact( system-module(?name, ?filename),
                    "Table of pairs of module names and system file names").

/*
** This table tells the description where to find the various system
** modules.
**
** It probably shouldn't embed the full directory path name
*/
:- assert( system-module("Binary", "Binary.def")),
   assert( system-module("Clock", "Clock.def")),
   assert( system-module("Convert", "Convert.def")),
   assert( system-module("ConvertLongReal", "ConvertLongReal.def")),
   assert( system-module("ConvertReal", "ConvertReal.def")),
   assert( system-module("Directory", "Directory.def")),
   assert( system-module("FileImplementation", "FileImplementation.def")),
   assert( system-module("FilePositions", "FilePositions.def")),
   assert( system-module("Files", "Files.def")),
   assert( system-module("InOut", "InOut.def")),
   assert( system-module("LongMathLib", "LongMathLib.def")),
   assert( system-module("LongRealIO", "LongRealIO.def")),
   assert( system-module("MathLib", "MathLib.def")),
   assert( system-module("NumberIO", "NumberIO.def")),
   assert( system-module("RealIO", "RealIO.def")),
   assert( system-module("SimpleIO", "SimpleIO.def")),
   assert( system-module("StandardIO", "StandardIO.def")),
   assert( system-module("Storage", "Storage.def")),
   assert( system-module("String", "String.def")),
   assert( system-module("System", "System.def")),
   assert( system-module("Terminal", "Terminal.def")),
   assert( system-module("Text", "Text.def")),
   assert( system-module("UnixDirectory", "UnixDirectory.def")),
   assert( system-module("UnixFiles", "UnixFiles.def")),
   assert( system-module("UnixParam", "UnixParam.def")),
   assert( system-module("UnixStdio", "UnixStdio.def")),
```

```
    assert( system-module("UnixTypes", "UnixTypes.def")).

import_seq
        =       import* => import_seq { }


definition_seq
        =       definition*         => definition_seq { }

/*
** Can-be-imported and Not-already-imported
**      are helper functions for the IMPORT operators
*/
can-be-imported(?name, ?entity, ?contour) :-
        not(<bound(?name, ?some-other-entity, ??), ?contour>) :
                ("~a is already bound in this scope", ?name) ,
        not-already-imported(?name, ?entity, ?contour).

not-already-imported(?name, ?entity, ?contour) :-
        <imported-name(?name, ?entity, ??), ?contour>.

not-already-imported(?name, ?entity, ?contour) :-
        not(<imported-name(?name, ?some-other-entity, ??), ?contour>).


import-enum-names(?enum-entity, ?text-loc, ?from-contour, ?to-contour) :-
        type-of(?enum-entity, enumeration-type(?enum-id-list)),
        !,
        for-all-children(:list, ?name, ?enum-id-list,
                { :all =>
                    <resolve-binding(?name, ?enum-ent, ??), ?from-contour>,
                    can-be-imported(?name, ?enum-ent, ?to-contour):
                        ("Unable to import enumeration constant ~a" ,
                                        ?name),
                    assert(imported-name(?name, ?enum-ent, ?text-loc), ?to-contour)
        }, ??, ??).
/* Succeed otherwise — record and pointer types carry their
** field with them. May have to add on here for arrays.
*/
import-enum-names(?enum-entity, ?text-loc, ?from-contour, ?to-contour).


import    =       "FROM" id "IMPORT" id_list ";" => qual_import {

        /*
        ** find-defn-module looks in global cm-controlled datapool for
        ** a definitions module. If it can't find it, and the
        ** module name is a system module, it attempts to load the
        ** system version.
        */
```

```
        find-defn-module(?name, ?mod-entity) :-
                global-cm-dp(?gdp),
                 <bound(?name, ?mod-entity, ??), ?gdp>.

        find-defn-module(?name, ?mod-entity) :-
                global-cm-dp(?gdp),
                system-module(?name, ?filename),
                lisp((load-modula2-file ?filename), ??),
                <bound(?name, ?mod-entity, ??), ?gdp>.

        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            pname($id, ?mod-name),
            find-defn-module(?mod-name, ?module-entity):
                        ("~a is not visible and so can't be imported from", ?mod-name),
            mode-of(?module-entity, module):
                ("~a is not a module name", ?mod-name),
            defn-contour(?module-entity, ?mod-contour),
            children($id_list, ?children),
            for-each-child(id, ?child, ?children,
                { :all => pname(?child, ?name),
                          <resolve-binding(?name, ?entity, ??), ?mod-contour> :
                                ("~a is not visible and so can't be imported", ?name),
                          can-be-imported(?name, ?entity, ?contour),
                          assert(imported-name(?name, ?entity, ?child), ?contour),
                          import-enum-names(?entity, ?child, ?mod-contour, ?contour)
                        }, ??, ??).
}

        |       "IMPORT" id_list ";" => rule_import {
        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            children($id_list, ?children),
            for-all-children(id, ?child, ?children,
                { :all => pname(?child, ?name),
                          find-defn-module(?name, ?entity):
                                ("~a is not visible and so can't be imported", ?name).
                          mode-of(?entity, module):
                                ("~a is not a module name", ?name),
                          can-be-imported(?name, ?entity, ?contour),
                          assert(imported-name(?name, ?entity, ?child), ?contour)
                }, ??, ??).
}

export  =       "EXPORT" id_list ";" => rule_export {
        //
```

```
        //
        :- context(?env),
            <contour(?contour), ?env>,
            children($id_list, ?children),
            for-all-children(id, ?child, ?children,
                  { :all => pname(?child, ?name),
                        local-binding(?child,?entity):
                              ("~a is not visible and so can't be exported", ?name),
                        assert(export-unqualified(?name, ?entity), ?contour)
                  }, ??, ??).
}
        |     "EXPORT" "QUALIFIED" id_list ";" => qual_export {
        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            children($id_list, ?children),
            for-all-children(id, ?child, ?children,
                  { :all => pname(?child, ?name),
                        local-binding(?child, ?entity) :
                              ("~a is not visible and so can't be exported", ?name),
                        assert(export-qualified(?name, ?entity), ?contour)
                  }, ??, ??).
}

opt_export
        =     [ export ]        => { }
```

Procedure Declarations

```
fact(returns(?result),
    "Asserted by a RETURN statement to show that value is returned from function").
fact(function-return-type(?type)).
fact(return(?type)).

scalar-type(?type-d) :-
        not(structured-type(?type-d)).

structured-type(array-type(??, ??)).
structured-type(record-type(??)).
structured-type(open-array()).
structured-type(set-type(??, ??, ??)).
structured-type(module-type(??)).
structured-type(named-type(?t)) :- structured-type(?t).

procedure_declaration
        =     procedure_heading block id ";" => procedure_declaration {
        datapool-name(proc-pool).
        context-name(proc-env).
```

```
        //
        :-  context(?context),
             <contour(?contour), ?context>,
             new-datapool(proc-pool, ?pool,
                          (contour-type(procedure(?pool)),
                           parent-contour(?contour),
                           open-contour())),
             new-context*(proc-env, ?newenv, ( contour(?pool))).
        //
          :-
             proc-name($procedure_heading, ?name1),
             pname($id, ?name1):
                    "Identifier at end of procedure does not match that at the beginning".
}

/* A procedure heading can appear either within a procedure, or within
** a module. Decl-contour hides this fact.
*/
decl-contour(?contour, ?res) :-
        <contour-type(module(??, ??)), ?contour>,
        =(?contour, ?res).

decl-contour(?contour, ?parent) :-
        <parent-contour(?contour), ?parent>.

procedure_heading
        =       "PROCEDURE" id opt_formal_parms ";" => procedure_heading {

        entity-name(proc-ent).
        proc-name($procedure_heading, ?name) :- pname($id, ?name).

        //
        //
        :- context(?env),
            pname($id, ?name),
            proc-formals($opt_formal_parms, ?formals),
            <contour(?contour), ?env>,
            decl-contour(?contour, ?def-contour),
            <declarable(?name), ?def-contour> :
                 ("~a is already declared", ?name),
            new-entity(proc-ent, ?pe,
                 (mode-of(?pe, procedure),
                 type-of(?pe, procedure-type(?formals)))),
            assert(bound(?name, ?pe, $procedure_heading), ?def-contour).

        :- context(?env),
            <contour(?contour), ?env>,
            decl-contour(?contour,?def-contour),
```

```
            if(<contour-type(module(?mod-ent, implementation)), ?def-contour>,
                defn-contour(?mod-ent, ?defn-module-contour);

                proc-ent(?proc-ent),
                pname($id, ?name),
                <bound(?name, ?def-entity, ?text-loc), ?defn-module-contour>,
                valid-redefinition(?name, ?proc-ent, ?def-entity); ).
}

function_declaration
        =       function_heading block id ";" => function_declaration {
        datapool-name(proc-pool).
        context-name(proc-env).

        //
        :-  context(?context),
            <contour(?contour), ?context>,
            new-datapool(proc-pool, ?pool,
                                (contour-type(function(?pool)),
                                 open-contour(),
                                 parent-contour(?contour))),
            new-context*(proc-env, ?newenv, ( contour(?pool))).

        //
        :- proc-name($function_heading, ?name1),
           pname($id, ?name1):
                  "Identifier at end of function does not match that at the beginning".

        /*  Pass down return type */
        :- proc-pool(?func-con),
           func-type($function_heading, ?res-type),
           assert(function-return-type(?res-type), ?func-con).

        /* constraint to make sure that return statement is present */
        :- proc-pool(?func-con),
           <returns(??), ?func-con>:
                "Function does not return any value!".
}

function_heading
        =       "PROCEDURE" id opt_formal_parms  ":" qual_id ";" => function_heading {
        proc-name($function_heading, ?name) :- pname($id, ?name).
        entity-name(proc-ent).

        func-type($function_heading, ?type) :-
                            qual-id-entity($qual_id, ?e),
                            type-descriptor(?e, ?type),
                            scalar-type(?type) : "Function return type must be scalar".
        //
        //
```

```
        :- context(?env),
            pname($id, ?name),
            proc-formals($opt_formal_parms, ?formals),
            func-type($function_heading, ?res-type),
            <contour(?contour), ?env>,
            decl-contour(?contour, ?def-contour),
            <declarable(?name), ?def-contour> :
                 ("~a is already declared", ?name),
            new-entity(proc-ent, ?pe,
                           (mode-of(?pe, function),
                            type-of(?pe, function-type(?formals, ?res-type)))),
            assert(bound(?name, ?pe, $function_heading), ?def-contour).

        :- context(?env),
            <contour(?contour), ?env>,
            decl-contour(?contour, ?def-contour),
             if(<contour-type(module(?mod-ent, implementation)), ?def-contour>,
                defn-contour(?mod-ent, ?defn-module-con);

                proc-ent(?proc-ent),
                pname($id, ?name),
                <bound(?name, ?def-entity, ?text-loc), ?defn-module-con>,
                valid-redefinition(?name, ?proc-ent, ?def-entity); ).
}

opt_formal_parms
        =          => empty_formal_parms {
        proc-formals($opt_formal_parms, []).
}
        |          "(" formal_parm_seq ")" => nonempty_formal_parms {
        proc-formals($opt_formal_parms, ?formals) :-
                children($formal_parm_seq, ?children),
                for-all-children(fp_section, ?child, ?children,
                     { :all =>
                          proc-formals(?child, ?formals-list)
                     },
                    ?formals-list, ?result-list) : "For all children fails!",
                /* result-list is a list of lists of entities. */
                lisp((flatten-list ?result-list), ?formals).
}


formal_parm_seq
        =          fp_section * ";" => formal_parm_seq {
}


/* Note—fp_section returns list of  entities  as proc-formals */
fp_section
        =          opt_var id_list ":" formal_type => fp_section
```

```
        {
        proc-formals($fp_section, ?entities) :-
                context(?env),
                <contour(?contour), ?env>,
                children($id_list, ?children),
                expr-type($formal_type, ?type),
                expr-type($opt_var, ?parm-type),
                for-each-child(id, ?child, ?children,
                { :all =>
                    pname(?child, ?name),
                    <declarable(?name), ?contour>:
                          ("Parameter ~a already declared in this scope", ?name),
                    new-entity(id-entity, ?e,
                                ( type-of(?e, ?type),
                                  mode-of(?e, ?parm-type))),
                    assert(bound(?name, ?e, ?child), ?contour)
                }, ?e, ?entities).
}

opt_var = "VAR"  => var_fp {
        expr-type($opt_var, var-parm).
}
        |        => val_fp {
        expr-type($opt_var, val-parm).
}
```

Type Declarations

```
/* Each of these has an "expr-type" property for the
** type that it represents.
*/
type
        =       simple_type         => { }
        |       array_type          => { }
        |       record_type         => { }
        |       set_type            => { }
        |       pointer_type        => { }
        |       procedure_type      => { }
        |       function_type       => { }

check-dbu(PREDEFINED,  ?nd).
check-dbu(?nd, PREDEFINED) :- !, fail().
check-dbu(?nd1, ?nd2) :- tree-ancestor(?nd1, ?nd2).

pure(check-dbu).

simple_type
        =       qual_id             => qual_id_type {

        expr-type($simple_type, ?e) :-
```

```
                qual-id-entity($qual_id, ?e),
                mode-of(?e, type) :  ("~a is not defined as a type!", ?e),
                owner(?e, ?nd) : "Owner fails~%",
                check-dbu(?nd, $simple_type) :
                        "Types must be defined before being used!".
        }
        |       enumeration      => { }
        |       subrange_type     => { }


selectable-contour(?name, ?entity, ?out-contour) :-
                type-descriptor(?entity, record(?contour)).

selectable-contour(?name, ?entity, ?out-contour) :-
                mode-of(?entity, module),
                defn-contour(?entity, ?out-contour).

<export-check(?name), ?contour> :-
                <record-contour(), ?contour>, !.

<export-check(?name), ?contour> :-
                <contour-type(module(??, ??)), ?contour>,
                <export-qualified(?name, ??, ??), ?contour>.

<export-check(?name), ?contour> :-
                <contour-type(module(??, ??)), ?contour>,
                <export-unqualified(?name, ??, ??), ?contour>.

qual_id
        =       id + "." => qual_id {


        qual-id-entity($qual_id, ?ent) :-
            context(?env),
            <contour(?contour), ?env>,
            children($qual_id, ?kids),
            /* This is a big iterator. Each node asserts an extrinsic node
            ** property "lpool" that is accessed by its right sib.
            ** The leftmost child uses the current contour as its starting
            ** point.
            */
            for-all-children(id, ?child, ?kids,
                { :singleton =>
                        pname(?child, ?name),
                        local-binding(?child, ?entity) :
                                ("~a is not bound here!", ?name)
                || /* first looks in contour */
                :first =>
                        pname(?child, ?name),
                        local-binding(?child, ?entity):
```

```
                        ("~a is not bound in this scope!", ?name),
                    selectable-contour(?name, ?entity, ?contour),
                    assert(lpool(?child, ?contour))
            || /* middle uses lpool */
            :middle =>
                    pname(?child, ?name),
                    left-sibling(?child, ?left-sib),
                    lpool(?left-sib, ?inh-contour),
                    <bound(?name, ?entity, ?text-loc), ?inh-contour>:
                        ("~a is not bound in this scope!", ?name),
                    export-check(?name, ?inh-contour),
                    selectable-contour(?name, ?entity, ?contour),
                    assert(lpool(?child, ?contour))
            || /* last uses lpool */
            :last =>
                    pname(?child, ?name),
                    left-sibling(?child, ?left-sib),
                    lpool(?left-sib, ?inh-contour),
                    export-check(?name, ?inh-contour),
                    <bound(?name, ?entity, ?text-loc), ?inh-contour>:
                        ("~a is not bound in this scope!", ?name)
            }, ?entity, ?result),
            lisp((car (last ?result)), ?ent).
    }


enumeration
        =       "(" id_list ")" => enumeration {
        entity-name(enum-entity).
        expr-type($enumeration, ?e) :- enum-entity(?e).
        //
        //
        :-  new-entity(enum-entity, ?e, (mode-of(?e, type))),
            context(?env),
            <contour(?contour), ?env>,
            children($id_list, ?kids),
            /* Collects the enum-ident-entity property for each
            ** child.
            */
            for-each-child(id, ?child,?kids,
                { :all =>
                        new-entity(id-entity,?id-entity,
                                ( type-of(?id-entity, ?e),
                                  mode-of(?id-entity, enum-constant(?id-entity, ?e)))),
                        pname(?child, ?name),
                        <declarable(?name), ?contour>:
                           ("Enum constant ~a already declared in this scope!", ?name),
                        assert(bound(?name, ?id-entity, ?child), ?contour)
                }, ?id-entity, ?enum-id-list),
             assert(type-of(?e, enumeration-type(?enum-id-list))).
```

```
}

in-subrange(subrange-type(??, ?min, ?max), ?v1, ?v2) :-
        <=(?min, ?v1),
        <=(?v1, ?max).

pure(in-subrange).


/* Introduce this maintained property to simplify the goals for
** the subrange_type operator.
*/
maintained-property(subrange-op-base-type).

subrange_type
        =       qual_id "[" const_expr ".." const_expr "]" => qual_subrange_type
        {
        entity-name(type-entity).

        subrange-op-base-type($subrange-type, ?base-type) :-
                    qual-id-entity($qual_id, ?qe),
                    type-of(?qe, ?qual-type),
                    base-type(?qual-type, ?base-type).

        expr-type($subrange_type, ?e) :-
            subrange-op-base-type($subrange_type, ?base-type),
            type-descriptor(?base-type, ?td),
            valid-subrange-type(?td),

            expr-mode($const_expr<1>, ?emode1),
            expr-mode($const_expr<2>, ?emode2),

            constant-value(?emode1, ?val1) :
                 "First value in subrange is not a constant",
            constant-value(?emode2, ?val2) :
                 "Second value in subrange is not a constant",

            <=(?val1, ?val2) :
                 "Second value in subrange exceeds first!",
            in-subrange(?td, ?val1, ?val2),
            new-entity(type-entity, ?e, ( mode-of(?e, type),
                                          type-of(?e, subrange-type(?base-type, ?val1, ?val2)))).
        //
        //
          :- subrange-op-base-type($subrange_type, ?base-type),
             expr-type($const_expr<1>, ?etype1),
             same-type(?base-type, ?etype1) :
                 "Type of first value in subrange is not the same as specified base type".
          :- subrange-op-base-type($subrange_type, ?base-type),
             expr-type($const_expr<2>, ?etype2),
```

```
            same-type(?base-type, ?etype2) :
                "Type of second value in subrange is not the same as specified base type".


        }
        |       "[" const_expr ".." const_expr "]" => unqual_subrange_type
        {
        entity-name(type-entity).

        expr-type($subrange_type, ?e) :-
            context(?env),
            <contour(?contour), ?env>,
            expr-mode($const_expr<1>, int-constant(?val1)) :
                "First value in subrange must be integral",
            expr-mode($const_expr<2>, int-constant(?val2)) :
                        "Second value in subrange must be integral",

            <=(?val1, ?val2) :
                "Second value in subrange exceeds first!",
            /*
            ** base type is INTEGER if first bound is negative, otherwise
            ** is CARDINAL!
            */
            if ( <(?val1, 0) ;
                <resolve-primitive("INTEGER", ?base-type, ??), ?contour> ;
                <resolve-primitive("CARDINAL", ?base-type, ??), ?contour>),
            type-descriptor(?base-type, ?td),
            in-subrange(?td, ?val1, ?val2),
            new-entity(type-entity, ?e, (mode-of(?e, type),
                                        type-of(?e, subrange-type(?base-type, ?val1, ?val2)))).

}

/* recurse down a list of expressions, getting type of the expression.
** On return, create nested type descriptors.
**
*/
make-array-type(?comp-type, [ ?index-e | ?rest ], array-type(?x, ?itype)) :-
                expr-type(?index-e, ?itype-e),
                type-descriptor(?itype-e, ?itype),
                valid-index-type(?itype),
                make-array-type(?comp-type, ?rest, ?x).

make-array-type(?comp-type, [ ?index-e ], array-type(?comp-type, ?itype)) :-
                expr-type(?index-e, ?itype-e),
                type-descriptor(?itype-e, ?itype),
                valid-index-type(?itype).

array_type
        =       "ARRAY" simple_type_list "OF" type => array_type {
```

```
        entity-name(type-entity).

        expr-type($array_type, ?e) :- type-entity(?e).
        //
        //
        :- expr-type($type, ?component-type),
           expr-type($simple_type_list, ?indices),
           make-array-type(?component-type, ?indices, ?array-descrip),
           new-entity(type-entity, ?e,
                        ( mode-of(?e, type),
                          type-of(?e, ?array-descrip))).
}

simple_type_list
        =       simple_type + "," => simple_type_list {
        expr_type($simple_type_list, ?type-list) :-
                children($simple_type_list, ?kids),
                for-all-children(simple_type, ?child, ?kids,
                        { :all => expr-type(?child, ?type) },
                        ?type, ?type-list).
}

pointer_type
        =       "POINTER" "TO" type => pointer_type
        {
        entity-name(type-entity).
        expr-type($pointer_type, ?e) :- type-entity(?e).
        //
        //
        :- expr-type($type, ?base-type),
           new-entity(type-entity, ?e, (mode-of(?e, type),
                                        type-of(?e, pointer-type(?base-type)))).
        }


procedure_type
        =       "PROCEDURE" function_type_seq => procedure_type
        {
        entity-name(type-entity).
        expr-type($procedure_type, ?e) :- type-entity(?e).
        //
        //
        :- expr-type($function_type_seq, ?formal-types),
           new-entity(type-entity, ?e, (mode-of(?e, type),
                                        type-of(?e, procedure-type(?formal-types)))).
        }

function_type
        =       "PROCEDURE" function_type_seq  ":" qual_id => function_type
        {
```

```
        entity-name(type-entity).
        expr-type($function_type, ?e) :- type-entity(?e).
        //
        //
        :- expr-type($function_type_seq, ?formal-types),
            qual-id-entity($qual_id, ?qe),
            type-of(?qe, ?res-type),
            scalar-type(?res-type),
            new-entity(type-entity, ?e, (mode-of(?e, type),
                                         type-of(?e, function-type(?formal-types, ?res-type)))).
        }

function_type_seq
        =           => empty_function_types
        {
        expr-type($function_type_seq, []).
        }
        |           "(" formal_function_type_seq ")"  => nonempty_function_types
        {
        /* Collect types of children into list */
        expr-type($function_type_seq, ?type-list) :-
                children($formal_function_type_seq, ?children),
                for-all-children(formal_function_type, ?child, ?children,
                        { :all => expr-type(?child, ?ctype)},
                        ?ctype, ?types).
        }

formal_function_type_seq
        =           formal_function_type * ";" => formal_function_type_seq { }


/*
** This code has to emulate the code for regular definitions.
** Thus, the type has to be a list of entities, each of which has a mode
** that is var-parm/val-parm and a type that is the type of the formal
** parameter.
*/
formal_function_type
        =           opt_var formal_type => formal_function_type
        {
        entity-name(formal-entity, "represents formal type in function type").

        expr-type($formal_function_type, ?e) :-
                expr-type($formal_type, ?type),
                expr-type($opt_var, ?parm-type),
                new-entity(formal-entity, ?e,
                                ( type-of(?e, ?type),
                                  mode-of(?e, ?parm-type))).
}
```

```
/* Returns type entity as expr-type */
/* the entity of the qualid is a type entity. */
formal_type
        =       qual_id => simple_formal_type {
        expr-type($formal_type, ?e) :-
                qual-id-entity($qual_id, ?e) : "Qual-id-entity fails!~%",
                mode-of(?e, type) :
                        "Qualified Identifier does not represent a type!".
        }
        |       "ARRAY" "OF" qual_id => array_formal_type
        {
        /* Here we have to create a new array type descriptor.
        ** And qual-id better represent a type!
        */
        entity-name(type-e, "Represents the newly-created formal array type").

        expr-type($formal_type, ?e) :-
                qual-id-entity($qual_id, ?type),
                mode-of(?type, type) :
                        "Qualified Identifier does not represent a type!",
                new-entity(type-e, ?e, ( mode-of(?e, type),
                                         type-of(?e, open-array(?type)))).
        }
```

Records and Sets

```
record_type
        =       "RECORD" field_list "END" => record_type {
        entity-name(type-entity).
        datapool-name(rec-pool).
        context-name(rec-context).

        expr-type($RecordType, ?e) :- type-entity(?e).

        //
        :-   context(?env),
             <contour(?contour), ?env>,
             <contour-type(?contype), ?contour>,
             new-datapool(rec-pool, ?pool, (record-contour(),
                                            open-contour())),
             new-context*(rec-context, ?newenv,
                          ( parent-contour(?contour),
                            contour-type(?contype))),
             new-entity(type-entity, ?e,
                 ( mode-of(?e, type),
                   type-of(?e, record-type(?contour)))).
}
```

```
field_list
        =       opt_field + ";" => field_list { }

opt_field
        =       [ field ]               => { }

field
        =       id_list ":" type => rule_field {
        //
        //
          :-
            context(?env),
            <contour(?contour), ?env>,
            expr-type($type, ?type),
            children($id_list, ?idents),
            for-each-child(id, ?id, ?idents,
                {:all => pname(?id, ?name),
                         <declarable(?name), ?contour>,
                         new-entity(id-entity, ?e,
                                    (mode-of(?e, variable),
                                     type-of(?e, ?type))),
                         assert(bound(?name, ?e, $field), ?contour) },
                ??, ??).
}

        |       "CASE" ":" qual_id "OF" variant_list opt_else_fields "END" => union {
        /* Create new context, but not new contour.
        ** Establish the type  of the qual_id, and pass it down as
        ** the case-label-type.
        */
                context-name(case-context).
                datapool-name(case-dp).
        //
        :-  context(?env),
            <contour(?contour), ?env>,
            <contour-type(?contype), ?contour>,
            new-datapool(case-dp, ?dp),
            new-context*(case-context, ?new-con,
                         (contour(?contour),
                          case-label-dp(?case-dp))).
        //
        :-
            qual-id-entity($qual_id, ?type),
            valid-index-type(?type),
            case-context(?context),
            assert(case-label-type(?type), ?context).

}
```

```
        |       "CASE" id ":" qual_id "OF" variant_list opt_else_fields "END" => discriminant
        {
        context-name(case-context).
        entity-name(field-ent).
        datapool-name(case-dp).
        //
        :-  context(?env),
            <contour(?contour), ?env>,
            <contour-type(?contype), ?contour>,
            new-datapool(case-dp, ?dp),
            new-context*(case-context, ?new-con,
                         ( contour(?contour),
                           case-label-dp(?dp))).

        //
        :-
            qual-id-entity($qual_id, ?type),
            valid-index-type(?type),
            case-context(?context),
            assert(case-label-type(?type), ?context).

        :-  context(?env),
            <contour(?contour), ?env>,
            pname($id, ?name),
            qual-id-entity($qual_id, ?type-e),
            type-of($qual_id, ?type),
            new-entity(field-ent, ?e,
                         ( mode-of(?e, variable),
                           type-of(?e, ?type))),
            assert(bound(?name, ?e, $field), ?contour).
}

variant_list
        =       opt_variant + "|" => variant_list { }


opt_variant
        =                                       => empty_variant { }
        | case_label_list ":" field_list  => variant
        {
        /*
        ** For code generation, would have to associate the cases in the
        ** label list with each field.
        **
        ** There are two ways to do this: either propagate the values in
        ** the case label list to the field declarations, or gather up
        **  the entities representing the fields and attach a list of
        ** case labels.
        ** The  first approach requires creating a new context at this
        ** point.
```

```
        ** The second approach requires that the entity for each field
        ** in the field_list be available.
        */
}

case_label_list
        =       case_label + "," => case_label_list {}

/*
** Not a node property, because it needs to be looked up in the database.
*/
fact(case-label-value(node, value1, value2, type)).
fact(case-label-dp(dp)).

/*
**
** Case labels get asserted into a special data pool.
**
** The function valid-case-label checks to see whether the declared value
** or range of values is already present in the database.
*/
valid-case-label(?dp, ?v1, ?v2) :-
                <case-label-value(??, ?w1, ?w2, ??), ?dp>,
                range-conflicts(?v1, ?v2, ?w1, ?w2).


/* tests for overlapping ranges, assuming that the given values are
** numeric. This means that we have to convert enumeration values
** to integer values.
*/
range-conflicts(?v, ?v, ?v, ?v).
range-conflicts(?v1, ?v2, ?w1, ?w2) :-          >=(?v1, ?w1), <=(?v1, ?w2).
range-conflicts(?v1, ?v2, ?w1, ?w2) :-          >=(?w1, ?v1), <=(?w1, ?v2).
pure(range-conflicts).

valid-const-range(?node1, ?node2, ?val1, ?val2, ?restype) :-
        expr-type(?node1, ?restype),
        expr-type(?node2, ?restype) :
            "Types of constants in subrange are not compatible!",
        expr-mode(?node1, ?mode1),
        expr-mode(?node2, ?mode2),
        constant-value(?mode1, ?val1) :
                "First value in subrange is not a constant!",
        constant-value(?mode2, ?val2) :
                "Last value in subrange is not a constant!",
        <=(?val1, ?val2) :
                "First value  exceed second value!".

/*
** subtype(A, B) :- A is a subtype of B
```

```
** Both same type
** constant-type is a subrange of label-type
*/
subtype(?type, ?type).
subtype(?type-a, ?type-b) :-
        type-of(type-a, subrange(?base, ??, ??)),
        same-type(?base, ?type-b).

case_label
        =       const_expr => case_element {
        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            expr-type($const_expr, ?type),
            expr-mode($const_expr, ?mode),
            constant-value(?mode, ?value),
            <case-label-type(?stype), ?env>,
            <case-label-dp(?label-dp), ?env>,
            subtype(?type, ?stype),
            valid-case-label(?label-dp, ?value, ?value),
            assert(case-label-value($case_label, ?value, ?value, ?type), ?label-dp).

}

        |       const_expr ".." const_expr => case_range {
        //
        //
          :- valid-const-range($const_expr<1>, $const_expr<2>, ?value1, ?value2, ?type),
              context(?env),
              <contour(?contour), ?env>,
              <case-label-type(?stype), ?env>,
              <case-label-dp(?label-dp), ?env>,
              subtype(?type, ?stype),
              valid-case-label(?label-dp, ?value1, ?value2),
              assert(case-label-value($case_label, ?value1, ?value2, ?type), ?label-dp).

}

opt_else_fields
        =       [ "ELSE" field_list ] => else_fields { }

case_stmt
        =       "CASE" expr "OF" case_list opt_else "END" => case_stmt {
        /* Create a new context, and pass down type of expr. */
        context-name(case-context).
        entity-name(field-ent).
        datapool-name(case-dp).
        //
        :- context(?env),
```

```
            <contour(?contour), ?env>,
            new-datapool(case-dp, ?dp),
            new-context*(case-context, ?new-con,
                         ( contour(?contour),
                           case-label-dp(?dp))).

        //
        :-
            expr-type($expr, ?type),
            valid-index-type(?type),
            case-context(?context),
            assert(case-label-type(?type), ?context).
}

case_list
        =       opt_case + "|" => case_list { }

opt_case
        /* Checking on validity of case labels is done in the constraints for
        ** case_label
        */
        =       [ case_label_list ":" stmt_list ] => case { }


with_stmt
        /* Need to create new contour with scope of designator  */
        /* Designator has to be a variable of type record-type(...) */
        /* this is a bit tricky—we propagate a new context, but we don't
        ** establish a new CONTOUR for it until the constraint evaluation
        ** phase. This, in turn, implies that other context-part constraint
        ** that use the current contour can  fail.
        */
        =       "WITH" designator "DO" stmt_list "END" => with_stmt
        {
        context-name(with-con).
        //
        :- context(?env),
            <contour(?contour), ?env>,
            <contour-type(?contype), ?contour>,
            new-context(with-con,?e, [$stmt_list],
                                     (parent-contour(?contour),
                                      contour-type(?contype))).

        //
        :- context(?env),
            <contour(?contour), ?env>,
            designator-entity($designator, ?des-ent),
            type-of(?des-ent, ?dtype),
            type-descriptor(?dtype, record-type(?field-scope)):
                 "Designator in with-statement is not a record!",
```

```
            with-con(?new-contour),
            assert(contour(?field-scope), ?new-contour).

}

set_type
        =       "SET" "OF" simple_type => set_type
        {
        entity-name(type-entity).
        expr-type($set_type, ?e) :- type-entity(?e).
        //
        //
        :- expr-type($simple_type, ?type),
            valid-set-type(?type),
            new-entity(type-entity, ?e,
                                        (mode-of(?e, type),
                                         type-of(?e, set-type(?type)))).
        }

fact(type-of-set(type)).
set_expr
        =           "{" element_seq "}" => undesig_set
        {
        context-name(set-expr-context).
        expr-type($set_expr, ?type-e) :-
            context(?env),
            <contour(?contour), ?env>,
            <resolve-primitive("BITSET", ?type-e, ??), ?contour>.

        expr-mode($set_expr, set-constant(?values)) :-
                children($element_seq, ?children),
                for-all-children(set_element, ?child, ?children,
                        { :all => expr-mode(?child,set-constant(?value))},
                        ?value, ?values).

        //
        :- context(?env),
            /* Calls own maintained property! */
            <contour(?contour), ?env>,
            expr-type($set_expr, ?type-e),
            new-context*(set-expr-context, ?e,
                                            ( type-of-set(?type-e),
                                              contour(?contour),
                                              open-contour())).

}
        |       designator "{" element_seq "}" => desig_set
        {
        /*
        ** designator must be valid set type
```

```
        ** pass type down
        */
        context-name(set-expr-context).
        expr-type($set_expr, ?type) :-
                designator-entity($designator, ?des-e),
                type-of(?des-e, ?type),
                valid-set-type(?type).

        expr-mode($set_expr, set-constant(?values)) :-
                children($element_seq, ?children),
                for-all-children(set_element, ?child, ?children,
                        { :all => expr-mode(?child,set-constant(?value))},
                        ?value, ?values).
        //
        :- context(?env),
           /* Calls own maintained property! */
           <contour(?contour), ?env>,
           expr-type($set_expr, ?type-e),
           new-context*(set-expr-context, ?e,
                                ( type-of-set(?type-e),
                                        contour(?contour),
                                        open-contour())).
}


element_seq
        =       element * "," => element_seq { }


subrange-base-type(named-type(?t), ?b) :- subrange-base-type(?t, ?b).
subrange-base-type(subrange-type(?t, ??, ??), ?b) :- subrange-base-type(?t, ?b).
subrange-base-type(?x, ?x).
pure(subrange-base-type).

valid-set-element(?set-type, ?elt-type, ?elt-value, ?contour) :-
        type-descriptor(?set-type, subrange(?type, ?low, ?high)),
        !,
        subrange-base-type(?type, ?base-type),
        <resolve-primitive("INTEGER", ?base-type, ??), ?contour>,
        <=(?low, ?elt-value),
        <=(?elt-value, ?high).

/* this handles enumerations */
valid-set-element(?set-type, ?set-type, ?element-value, ?contour).

element
        =       const_expr          => set_element
        {
        expr-mode($set_expr, set-constant([?value])) :-
           context(?env),
```

```
            <contour(?contour), ?env>,
            <type-of-set(?type-e), ?env>,
            expr-type($const_expr, ?etype),
            expr-mode($const_expr, ?emode),
            constant-value(?emode, ?value),
            valid-set-element(?type-e, ?etype, ?value, ?contour).
}
        |       const_expr ".." const_expr => set_range {
        expr-mode($set_expr, set-constant([?value1, ?value2])) :-
            context(?env),
            <contour(?contour), ?env>,
            <type-of-set(?type-e), ?env>,
            expr-type($const_expr<1>, ?etype1),
            expr-mode($const_expr<1>, ?emode1),
            constant-value(?emode1, ?value1),
            valid-set-element(?type-e, ?etype1, ?value1, ?contour),

            expr-type($const_expr<2>, ?etype2),
            expr-mode($const_expr<2>, ?emode2),
            constant-value(?emode2, ?value2),
            valid-set-element(?type-e, ?etype2, ?value2, ?contour).
}


set-member(?v1, [], FALSE).
set-member(?v1, [ ?v1 | ?? ], TRUE).
set-member(?v1, [ ?hd | ?tl ], ?res) :- set-member(?v1, ?tl, ?res).
pure(set-member).

set_member_expr   =       expr "IN" expr            => member_expr {
        expr-type($set_member_expr,  ?bool-e) :-
            expr-type($expr<1>, ?t1),
            expr-type($expr<2>, set-type(?t1)),
            context(?env),
            <contour(?contour), ?env>,
            <resolve-primitive("BOOLEAN",?bool-e, ??), ?contour>.

        expr-mode($set_member_expr, bool-const(?res)) :-
                expr-mode($expr<1>, ?m1),
                constant-value(?m1, ?v1),
                expr-mode($expr<2>, ?m2),
                constant-value(?m2, set-const(?v2)),
                set-member(?v1, ?v2, ?res).

        expr-mode($set_member_expr, expression).
}
```

Declarations

definition

```
        =       "CONST" constant_declaration_seq => const_definition { }
        |       "TYPE" type_definition_seq => rule_type_definition { }
        |       "VAR" var_declaration_seq => var_definition { }
        |       procedure_heading  => { }
        |       function_heading  => { }


constant_declaration_seq
        =       constant_declaration* => constant_declaration_seq { }


type_definition_seq
        =       type_definition* => type_definition_seq { }


var_declaration_seq
        =       var_declaration* => var_declaration_seq { }

/*
** The next procedures are used in ensuring that declarations that
** appear in both an implementation module and a definition module
** are the same.
*/
valid-redefinition(?name, ?impl-ent, ?def-ent) :-
        mode-of(?impl-ent, ?mode),
        mode-of(?def-ent, ?mode) : ("Invalid redefinition of ~a", ?name),
        type-of(?impl-ent, ?type1),
        type-of(?def-ent, ?type2),
        same-type(?type1, ?type2) :
                ("Invalid redefinition of ~a (type mismatch)", ?name).

valid-type-redefinition(?name, ?impl-ent, ?def-ent) :-
        type-of(?def-ent, opaque-type),
        type-of(?impl-ent, pointer-type(??)) :
                ("Opaque type ~a must be implemented using a pointer type!", ?name).

valid-type-redefinition(?name, ?impl-ent, ?def-ent) :-
        valid-redefinition(?name, ?impl-ent, ?def-ent).

declaration
        =       "CONST" constant_declaration_seq => rule_declaration { }
        |       "TYPE" type_declaration_seq => rule_declaration { }
        |       "VAR" var_declaration_seq =>  rule_declaration { }
        |       procedure_declaration  => { }
        |       function_declaration  => { }
        |       module_declaration  => { }


type_declaration_seq
        =       type_declaration* => type_declaration_seq { }
```

```
constant_declaration
        =       id "=" const_expr ";" => const_declaration {
                entity-name(const-entity).
        //
        //
        /*
        ** Note: a constant expression has a value of a form suitable to be
        ** a constant descriptor.
        **
        */
         :-  pname($id, ?name),
             context(?env),
             <contour(?contour), ?env>,
             <declarable(?name), ?contour> : ("~a is already bound", ?name),
             expr-type($const_expr, ?expr-type),
             expr-mode($const_expr, ?mode),
             new-entity(const-entity, ?e,
                 (mode-of(?e, ?mode),
                  type-of(?e, ?expr-type))),
             assert(bound(?name, ?e, $constant_declaration), ?contour).

        :- context(?env),
           <contour(?contour), ?env>,
           <contour-type(module(?mod-ent, implementation)), ?contour>,
           defn-contour(?mod-ent, ?def-con),
           pname($id, ?name),
           <bound(?name, ?def-entity, ?text-loc), ?def-con>,
           const-entity(?const-ent),
           valid-redefinition(?name, ?const-ent, ?def-entity).
        }

type_declaration
        =       id "=" type ";" => type_declaration {
        entity-name(type-entity).
        //
        //
         :-
             context(?env),
             pname($id, ?name),
             expr-type($type, ?expr-type),
             <contour(?contour), ?env>,
             <declarable(?name), ?contour> :
                 ( "~a is already bound", ?name),
             new-entity(type-entity, ?e,
                          ( mode-of(?e, type),
                            owner(?e, $type_declaration),
                            type-of(?e, ?expr-type))),
             assert(bound(?name, ?e, $type_declaration), ?contour).
```

```
        :-  context(?env),
            <contour(?contour), ?env>,
            <contour-type(module(?mod-ent, implementation)), ?contour>,
            defn-contour(?mod-ent, ?def-con),
            type-entity(?type-ent),
            pname($id, ?name),
            <bound(?name, ?def-entity, ?text-loc), ?def-con>,
            valid-type-redefinition(?name, ?type-ent, ?def-entity).
}

type_definition
        =       type_declaration   => { }
        |       id ";"                      => opaque_type_definition
        {
        entity-name(type-entity).
        //
        //
        :- context(?env),
           <contour(?contour), ?env>,
           pname($id, ?name),
           <declarable(?name), ?contour>,
           new-entity(type-entity, ?e,
                                ( mode-of(?e, type),
                                  owner(?e, $type_definition),
                                  type-of(?e, opaque-type))),
          assert(bound(?name, ?e, $type_definition), ?contour).
}

/*
**
*/
var_declaration
        =       id_list ":" type ";" => var_declaration {
        //
        //
          :-
              context(?env),
              <contour(?contour), ?env>,
              expr-type($type, ?expr-type) : "Expr-type failed!~%",
              children($id_list, ?children),
              for-each-child(id, ?node, ?children,
                        { :all =>  pname(?node, ?name),
                                <declarable(?name), ?contour> :
                                        ( "~a is already bound here", ?name),
                                new-entity(id-entity, ?e, ( mode-of(?e, variable),
                                                            type-of(?e, ?expr-type))),
                                assert(bound(?name, ?e, ?node), ?contour)
                        }, ??, ??).
```

```
        :-  context(?env),
            <contour(?contour), ?env>,
            if( <contour-type(module(?mod-ent, implementation)), ?contour>,
                defn-contour(?mod-ent, ?def-con);
                children($id_list, ?children),
                for-each-child(id, ?node, ?children,
                { :all =>
                        id-entity(?var-ent),
                        pname(?node, ?name),
                         <bound(?name, ?def-entity, ?text-loc), ?def-con>,
                        valid-redefinition(?name, ?var-ent, ?def-entity)
                }, ??, ??) ;
                ) : "Recheck of definitions fails!~%".
        }


id_list
        =       id +"," => id_list { }
```

Expressions

```
expression = expr => expression {
        expr-type($expression, ?type) :- expr-type($expr, ?type).
        expr-mode($expression, ?mode) :- expr-mode($expr, ?mode).
}

expr
        =       expr relop expr             => relational_expr {
        expr-type($expr<0>, ?res) :-
            expr-type($expr<1>, ?t1),
            expr-type($expr<2>, ?t2),
            expr-mode($expr<1>, ?m1),
            expr-mode($expr<2>, ?m2),
            pname($relop, ?op),
            expression-compatible(?t1, ?m1, ?t2, ?m2, ?expr-type) :
                "Expression components are not type compatible~%",
            binop-base-type(?expr-type, ?type-a),
            type-descriptor(?type-a, ?type-d),
            binop-compatible(?type-d, ?op) :
                (" ~a is not binop-compatible with ~a~%", ?type-d, ?op),
            context(?env),
            <contour(?contour), ?env>,
            <resolve-primitive("BOOLEAN", ?res, ??), ?contour>.

          expr-mode($expr, ?mode) :-
            expr-mode($expr<1>, ?m1),
            expr-mode($expr<2>, ?m2),
            constant-value(?m1, ?v1),
            constant-value(?m2, ?v2),
```

```
        pname($relop, ?op),
        eval-binop(?op, ?m1, ?v1, ?m2, ?v2, ?mode).

    /*
    ** If constant evaluation fails, just default mode to expr.
    */
      expr-mode($expr,  expression).
}
    |       expr addop expr          => binary_expr { }
    /* This definition works for all binary_exprs */
    |       expr mulop expr          => binary_expr {
expr-type($expr<0>, ?res) :-
    expr-type($expr<1>, ?t1),
    expr-type($expr<2>, ?t2),
    expr-mode($expr<1>, ?m1),
    expr-mode($expr<2>, ?m2),
    /* Can just use pname here, since the operator token will be
    ** the second child of the node
    */
    pname($mulop, ?op),
    expression-compatible(?t1, ?m1, ?t2, ?m2, ?res) :
        "Expression components are not type compatible~%",
    binop-base-type(?res, ?type-a),
    type-descriptor(?type-a, ?type-d),
    binop-compatible(?type-d, ?op) :
        (" ~a is not binop-compatible with ~a~%", ?type-d, ?op) .

  expr-mode($expr, ?mode) :-
    expr-mode($expr<1>, ?m1),
    expr-mode($expr<2>, ?m2),
    constant-value(?m1, ?v1),
    constant-value(?m2, ?v2),
    pname($mulop, ?op),
    eval-binop(?op, ?m1, ?v1, ?m2, ?v2, ?mode).

    /*
    ** If constant evaluation fails, just default mode to expr.
    */
      expr-mode($expr,  expression).
}

    |       "+" expr                 => unary_plus_expr {
expr-type($expr<1>, ?type) :-
    expr-type($expr<1>, ?type),
    unop-compatible(?type, "+").

expr-mode($expr<0>, ?mode) :-
    expr-mode($expr<1>, ?mode).
expr-mode($expr<0>, expression).
}
```

```
        |       "-" expr                    => unary_minus_expr
        {
        expr-type($expr<1>, ?type) :-
            expr-type($expr<1>, ?type),
            unop-compatible(?type, "-"):
            ("~a is not compatible with a unary minus", ?a).

        expr-mode($expr<0>, int-constant(?nv1)) :-
            expr-mode($expr<1>, ?m1),
            constant-value(?m1, ?v1),
            /* Assumes valid integer here! */
            lisp((- ?v1), ?nv1).
        expr-mode($expr<0>, expression).
}
        |       integer             => int_const {
        expr-type($expr, ?entity) :-
                context(?env),
                <contour(?contour), ?env>,
                <resolve-primitive("INTEGER", ?entity, ??), ?contour>.
        expr-mode($expr, int-constant(?value)) :-
                pname($integer, ?name),
                atoi(?name, ?value).
}
        |       real                => real_const {
        expr-type($expr, ?entity) :-
                context(?env),
                <contour(?contour), ?env>,
                <resolve-primitive("REAL", ?entity, ??), ?contour>.
        expr-mode($expr, real-constant(?value)) :-
                pname($real, ?name),
                atof(?name, ?value).
}
        |       string              => string_const {
        expr-type($expr, ?entity) :-
                expr-type($string, ?entity).
        expr-mode($expr, ?mode):-
                expr-mode($string, mode).
}

        |       set_member_expr => { }
        |       set_expr => { }
        |       designator => proc_ref {
        /*
        ** If designator is constant, then so is result. Otherwise mode of
        ** result is set to "variable". The type field is simply passed up.
        */
        expr-mode($expr, ?des-mode) :-
                designator-entity($designator, ?des-ent),
                mode-of(?des-ent, ?des-mode).
        expr-mode($expr, expression).
```

```
        expr-type($expr, ?res-type) :-
                designator-entity($designator, ?des-ent),
                type-of(?des-ent, ?res-type).

}
        |       designator actual_parms     => fun_call {
        expr-mode($expr, expression).
        expr-type($expr, ?res-type) :-
                 context(?env),
                 <contour(?contour), ?env>,
                 designator-entity($designator, ?des-ent),
                 mode-of(?des-ent, ?des-mode): "Illegal function call",
                 type-of(?des-ent, function-type(?formals, ?res-type)),
                 actual-parms($actual_parms, ?actuals),
                 actuals-match(?contour, ?actuals, ?formals) :
                         "actual parameters do not match definition".
}
        |       "NOT" expr                  => not {
          expr-mode($expr<0>, ?new-mode) :-
            expr-mode($expr<1>, ?mode),
            eval-unop("NOT", ?mode, ?new-mode).

          expr-type($expr<0>,  BOOLEAN) :-
            expr-type($expr<1>, ?type),
            context(?env),
            <contour(?contour), ?env>,
            <resolve-primitive("BOOLEAN", ?b-type, ??), ?contour>,
            same-type(?type, ?b-type).
}
        |       "(" expr ")" => ANNOTATE {
        expr-mode($expr<0>, ?mode) :-
                expr-mode($expr<1>, ?mode).
        expr-type($expr<0>, ?type) :-
                expr-type($expr<1>, ?type).
 }


relop   =       "="     => { }
        |       "#"     => { }
        |       "<"     => { }
        |       "<="    => { }
        |       ">"     => { }
        |       ">="    => { }
        |       "<>"    => { }


addop
        =       "+" => { }
        |       "-" => { }
```

```
        |       "OR" => { }

mulop
        =        "*" => { }
        |        "/"=> { }
        |        "DIV" => { }
        |       "MOD" => { }
        |       "AND" => { }


/*
** Designators are interesting. The leftmost child of the sequence needs to
** know the entity bound to the leftmost "id". Otherwise, type and
** mode information can be passed left-to-right.
**
** Thus the designator should pass a new context to the des_sequence
** that contains the entity of the  leftmost child. But context propagation
** can't fail while a lookup on the "id" might very well fail.
** So instead, we pass the name of the id! Then, the leftmost child
** of the "des_seq" can use the node reference to get the entity
** of the id!
*/

designator
        =       id des_seq => designator {
        context-name(new-con).

        designator-entity($designator, ?ent) :-
                        designator-entity($des_seq, ?ent).

        designator-entity($designator, ?ent) :-
                        local-binding($id, ?ent).

        // Second-Pass Goals
        :- context(?env),
            <contour(?contour), ?env>,
            pname($id, ?name),
            new-context(new-con, ?nc, [$des_seq],
                        ( designator-head(?name),
                          contour(?contour) )).
}


des_seq
        =       des* => des_seq {
        /* The entity associated with a des_seq is the entity of its
        ** rightmost element. fails if the sequence is empty.
        */
```

```
        designator-entity($des_seq, ?e) :-
                children($des_seq, ?kids),
                lisp((car (last ?kids)), ?last-kid),
                not(=(?last-kid, nil)),
                designator-entity(?last-kid, ?e) .
}

/*
** Semantic checks for "des" have to have two alternatives—if this child
** is the leftmost child, then it needs to get its contextual information
** partly from the "designator-head" fact in the current context.
** Otherwise, it has to get it's information from its leftmost-sibling.
*/

get-designator-context(?self, ?env, ?entity) :-
                is-leftmost-child(?self), !,
                <designator-head(?name), ?env>,
                <contour(?contour), ?env>,
                <resolve-binding(?name, ?entity, ??), ?contour>.

get-designator-context(?self, ?env, ?entity) :-
                left-sibling(?self, ?left),
                designator-entity(?left, ?entity).

/* Select-properties does the work for a selector.
** The first clause succeeds only for records.
** If the procedure succeeds, it returns the type and mode
** information needed.
**
** There is no need to check the mode here, because we don't have
** record or module constants.
*/
selector-properties(?name, record-type(?scope), ?entity) :- !,
                <bound(?name, ?entity, ??), ?scope> :
                        ("?a is not a valid record field", ?name).

selector-properties(?name, module-type(?scope), ?entity) :- !,
                /* Check in module contour and all imports, all the
                ** way back. Has to be export-qualified from
                ** the given contour
                */
                <resolve-binding(?name, ?entity, ??), ?scope>:
                                        ("~a is not visible", ?name),
                <export-qualified(?name, ?entity), ?scope>:
                                        ("~a has not exported", ?name).
des
        =       "." id                  => selector {

        designator-entity($des, ?entity) :-
            context(?env),
```

```
            get-designator-context($des, ?env, ?l-entity),
            pname($id, ?name),
            type-of(?l-entity, ?ltype),
            selector-properties(?name, ?ltype, ?entity).
        }

        |       "[" expr_list "]"           => index
        {
        /* Have to create a new, partially dereference array entity
        ** here, since we're not passing both the type and mode.
        */
        entity-name(des-entity).

        designator-entity($des, ?de) :-
            context(?env),
            <contour(?contour), ?env>,
            children($expr_list, ?children),
            for-all-children(expression, ?expr, ?children,
                  { :all => expr-type(?expr, ?type),
                            expr-mode(?expr, ?mode)
                  }, [ ?type, ?mode ], ?expr-info-list),
            get-designator-context($des, ?env, ?l-entity),
             deref-array(?contour, ?l-entity,  ?expr-info-list, ?new-type),
             new-entity(des-entity, ?de,
                           ( mode-of(?de, variable),
                             type-of(?de, ?new-type))).
        }

        |       "^"                         => deref {
        /* Have to create a new entity here for the dereferenced  object.
        */
        entity-name(des-entity).
        designator-entity($des, ?de) :-
            context(?env),
            get-designator-context($des, ?env, ?l-entity),
            type-descriptor(?l-entity, pointer(?type)),
            mode-of(?l-entity, ?mode),
            new-entity(des-entity, ?de,
                           ( mode-of(?de, variable),
                             type-of(?de, ?type))).
}

/* Synthesizes "parms" */
maintained-property(actual-parms).
actual_parms
        =       "("  ")" => empty_actual_parms {
        actual-parms($actual_parms, []).
}
        |        "(" expr_list ")" => nonempty_actual_parms
        {
```

```
        actual-parms($actual_pars, ?expr-info-list) :-
            children($expr_list, ?children),
            for-all-children(expression, ?expr, ?children,
                  { :all => expr-type(?expr, ?type),
                            expr-mode(?expr, ?mode) },
                  [?type, ?mode], ?expr-info-list).
}

expr_list
        =       expr + "," => expr_list { }


const_expr
        =       expr => const_expr {
        expr-type($const_expr, ?type) :-    expr-type($expr, ?type).

        expr-mode($const_expr, ?mode) :-
             expr-mode($expr, ?mode),
             constant-value(?mode, ??) :  "Constant expression must be a constant!".
}


string ==> {
        entity-name(type-entity).

        expr-type($string, ?e) :- type-entity(?e).
        expr-mode($string, ?mode) :- pname($string, ?name),
                                        =(?mode, string-constant(?name)).

        //
        :- new-entity(type-entity, ?e,
                                ( mode-of(?e, type) )).

        //
        :- context(?env),
            <contour(?contour), ?env>,
            <resolve-primitive("CHAR", ?base-type, ??), ?contour>,
            <resolve-primitive("CARDINAL", ?card-type, ??), ?contour>,
            pname($string, ?name),
            type-entity(?e),
            lisp((string-length ?name), ?len),
            assert(type-of(?e, array-type(?base-type,
                                [ subrange( ?card-type, 0, ?len) ]))).

}
```

Statements

```
block
        =       decl_seq "END"                          => decl_block { }
        |       decl_seq "BEGIN" stmt_list "END"        => rule_block { }

decl_seq
        =       declaration* => decl_seq { }

stmt_list
        =       opt_stmt + ";" => stmt_list { }

opt_stmt
        =       [ stmt ]            => { }

stmt
        =       assignment          => /* ANNOTATE */ { }
        |       proc_call           => /* ANNOTATE */ { }
        |       if_stmt             => /* ANNOTATE */ { }
        |       case_stmt           => /* ANNOTATE */ { }
        |       while_stmt          => /* ANNOTATE */ { }
        |       repeat_stmt         => /* ANNOTATE */ { }
        |       loop_stmt           => /* ANNOTATE */ { }
        |       for_stmt            => /* ANNOTATE */ { }
        |       with_stmt           => /* ANNOTATE */ { }
        |       "EXIT"              => exit_stmt { }
        /* Have to put presence of returns into local scope to assure taht
        ** they exist.
        */
        |       "RETURN"            => void_return
        {
        /* Must be in procedure or module body */
        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            not(<contour-type(function), ?contour>) :
                "Return inside a function must specify a value!".
}
        |       "RETURN" expression         => expr_return {
        /* Must be in function body
        ** Type of expr must be assignment compatible with function
        ** result.
        */
        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            <contour-type(function(?fcon)), ?contour>:
                "Can only return values from functions!",
            expr-mode($expression, ?emode),
            expr-type($expression, ?etype),
```

```
            <function-return-type(?res-type), ?fcon>,
            assignment-compatible(?contour, ?res-type, expression, ?etype, ?emode):
                  "Return type of function not compatible with expression!",
            /* Assert value here that return statement exists */
            assert(returns(?etype), ?fcon).
}

assignment
        =       designator ":=" expression => assignment {

        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            designator-entity($designator, ?des-ent),
            mode-of(?des-ent, ?mode),
            assignable-loc(?mode): ("Can't assign to a ~a", ?mode),
            expr-type($expression, ?expr-type),
            expr-mode($expression, ?expr-mode),
            type-of(?des-ent, ?des-type),
            assignment-compatible(?contour, ?des-type, ?mode, ?expr-type, ?expr-mode) :
                  ("Assigning a ~a of type ~a to a ~a of type ~a is illegal",
                        ?expr-mode, ?etype, ?mode, ?des-type).

}

proc_call
        =       designator                  => proc_call_no_parms {
        /* Check that designator is a procedure of 0 params */
        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            designator-entity($designator, ?des-ent),
            mode-of(?des-ent, procedure) :
                  "Illegal procedure call",
            type-of(?des-ent, procedure-type([])) :
                  "Arguments expected for this procedure!".
}
        |       designator actual_parms => rule_proc_call {
        /* Check that designator is a procedure of N params */
        //
        //
        :- context(?env),
            <contour(?contour), ?env>,
            designator-entity($designator, ?des-ent),
            mode-of(?des-ent, procedure) :
                  "Illegal procedure call",
            type-of(?des-ent, procedure-type(?formals)),
            actual-parms($actual_parms, ?actuals),
```

```
            actuals-match(?contour, ?actuals, ?formals) :
                    "actual parameters do not match definition".
}


<valid-boolean-expression(?type), ?contour> :-
        <resolve-primitive("BOOLEAN", ?system-bool, ??), ?contour>,
        type-descriptor(?type, ?type-d),
        same-type(?type, ?system-bool).

if_stmt
        =       "IF" expression "THEN" stmt_list elsif_seq opt_else "END" => if_stmt
        {
        //
        //
        :- expr-type($expression, ?type),
           context(?env),
           <contour(?contour), ?env>,
           <valid-boolean-expression(?type), ?contour> :
                "Type of expression is not system's boolean!%".
}

elsif_seq
        =       elsif* => elsif_seq { }

elsif
        =       "ELSIF" expression "THEN" stmt_list => elsif {
        //
        //
        :- expr-type($expression, ?type),
           context(?env),
           <contour(?contour), ?env>,
           <valid-boolean-expression(?type), ?contour> :
                "Type of expression is not system's boolean!%".
}

opt_else
        =        [ "ELSE" stmt_list ] => else { }

while_stmt
        =       "WHILE" expression "DO" stmt_list "END" => while_stmt {
        //
        //
        :- expr-type($expression, ?type),
           context(?env),
           <contour(?contour), ?env>,
           <valid-boolean-expression(?type), ?contour> :
                "Type of expression is not system's boolean!".
}
```

```
repeat_stmt
        =       "REPEAT" stmt_list "UNTIL" expression => repeat_stmt {
        //
        //
        :- expr-type($expression, ?type),
            context(?env),
            <contour(?contour), ?env>,
            <valid-boolean-expression(?type), ?contour> :
                "Type of expression is not system's boolean!".
}

for_stmt
        =       "FOR" id ":=" expression "TO" expression opt_by "DO" stmt_list "END"
                /* Id a variable, id, expr1, assignment compatible
                ** expr1, expr2 exrpession compatible
                */

        => for_stmt {
        qual-id-entity($for_stmt, ?entity) :-
            context(?env),
            <contour(?contour), ?env>,
            pname($id, ?id-name),
            <resolve-binding(?id-name, ?id-entity, ??), ?contour>:
                ("Loop control variable ~a is not declared", ?name).

        //
        //
        :- qual-id-entity($for_stmt, ?id-entity),
            mode-of(?id-entity, ?id-mode),
            =(?id-mode, variable) :
                "Loop control variable must be assignable".

        :- qual-id-entity($for_stmt, ?id-entity),
            context(?env),
            <contour(?contour), ?env>,
            mode-of(?id-entity, ?id-mode),
            type-of(?id-entity, ?id-type),
            expr-type($expression<1>, ?be-type),
            expr-mode($expression<1>, ?be-mode),
            assignment-compatible(?contour, ?id-type, ?id-mode, ?be-type, ?be-mode):
                "Loop control variable and base expression are not compatible".

        :-
            expr-type($expression<1>, ?be-type),
            expr-mode($expression<1>, ?be-mode),
            expr-type($expression<2>, ?ee-type),
            expr-mode($expression<2>, ?ee-mode),
            expression-compatible(?be-type, ?be-mode, ?ee-type, ?ee-mode, ??):
                "Lower and upper bounds of loop are not compatible expressions".
}
```

```
opt_by
        =                                   => empty_by { }
        |       "BY" const_expr             => by_value {
        //
        //
        :- expr-mode($const_expr, int-constant(?val)):
            "Loop by-value must be integral!".
}

loop_stmt
        =       "LOOP" stmt_list "END" => loop_stmt { }
```

Index