1

CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Division

Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

60003

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

CHRISTOPHER ANTHONY CARTER

Date of Birth — Date de naissance

AUGUST 6, 1955

Country of Birth — Lieu de naissance

CANADA

Permanent Address — Résidence fixe

3 GREENHILL WAY
OTTAWA ONTARIO
K1K 0R5

Title of Thesis — Titre de la thèse

THE DESIGN OF A PROGRAM EDITOR BASED
ON CONSTRAINTS

University — Université

CARLETON UNIVERSITY

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF SCIENCE

Year this degree conferred — Année d'obtention de ce grade

1983

Name of Supervisor — Nom du directeur de thèse

WILF R. LALONDE

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHEQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur

Date

OCTOBER 17, 1983

Signature

Chris Carter

THE DESIGN OF A PROGRAM EDITOR BASED ON CONSTRAINTS

by

© Christopher A. Carter, B.Sc.

A thesis submitted to the the Faculty of
Graduate Studies and Research in partial fulfilment
of the requirements for the degree of

Master of Science

Department of Mathematics and Statistics

Carleton University

Ottawa, Ontario

September 30, 1983

The undersigned recommend to the Faculty of Graduate
Studies and Research acceptance of the thesis

THE DESIGN OF A PROGRAM EDITOR BASED ON CONSTRAINTS

submitted by Christopher A. Carter, B.Sc.
in partial fulfilment of the requirements for
the degree of Master of Science.

THESIS SUPERVISOR

CHAIRMAN, DEPARTMENT OF MATHEMATICS AND STATISTICS

Carleton University

October 14, 1983

ABSTRACT

This thesis describes a program editor based on the constraint system developed by Steele and Sussman. Constraints are shown to be a particularly good formalism for specifying the syntactic and static semantic rules of a programming language. By embedding the abstract syntax tree of a program into a constraint network, the context dependent rules of the language can be enforced by the propagation of information through the network.

Since propagation (and retraction) of information occurs automatically when network fragments are connected (or disconnected), incremental editing is easily achieved. The ability of the system to record the history of computation allows the editor to provide good explanations of (and to recover from) user errors.

ACKNOWLEDGEMENTS

Joe Catastrophe, Joe Tuba and I would like to give our thanks to all of those wonderful creatures of the cosmos who in some way, large or small, contributed to this thesis. To my supervisor Wilf LaLonde who somehow had the patience to put up with my turtle's rate of progress. To Jim desRivieres who gave me enough encouragement way back there in the Pre-Cambrian Era (the fall of 1981) to make me believe it could actually be done. To Dave Thomas who graciously allowed a slight case of absenteeism from a Dynasty in the quest for this holy grail. To Wilf, Jim and Dave, I thank you for all of the discussions and inspiration which led to many of the ideas presented here.

To all of my friends: Susan Ascroft and Tersh Doe, Duncan Glendinning, Paul Matthews, Barbara and Glen Ford, and Frank and Monika Plettenberg-Smith. Your constant pokes and jokes were very much appreciated; they never let me lose sight of my objective.

To David Kennedy and John Ford who kept me racing around the Gatineau Hills on a bicycle when I really should have been working on this thesis.

To my fellow strugglers in the Information and Systems Science program: Kathleen Byrne and Ian Gilchrist. Thanks for the company.

To G. B. Trudeau and Walt Kelly. Thanks for the laughs.

To Mr., Mrs. and Margie Snarr, thanks a bazillion for your abundant hospitality and encouragement.

To the Carter family whose faith in me never wavered. You have given me all the support and love one could ever ask for. (And thanks for the pictures, Joe!)

And finally to Victoria Snarr. Without you, this work would never have been possible. You have been my editor-in-chief, my proofreader, my typist, my critic, my teacher, my consultant and above all my constant companion and friend. All of this at a time when you had your own thesis to work on. To you Vicky, I dedicate this thesis.

CONTENTS

	Page
ACCEPTANCE	ii
ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
1. INTRODUCTION.	1-1
1.1 A Short Story.	1-1
1.2 Inadequate Programming Environments.	1-2
1.3 Related Research	1-5
1.3.1 Teletype text editors	1-5
1.3.2 Teletype program editors.	1-8
1.3.3 Screen text editors	1-15
1.3.4 Screen program editors.	1-20
1.3.5 Knowledge based editors	1-24
1.4 Wouldn't It Be Nice?	1-26
1.5 Thesis Overview.	1-28
2. CONSTRAINTS	2-1
2.1 Another Short Story.	2-1
2.2 Introduction To Constraints.	2-3
2.2.1 Propagation	2-7
2.2.2 Dependencies.	2-13
2.2.3 Retraction.	2-16
2.3 Constraints And Program Editors.	2-25
3. EDITOR OVERVIEW	3-1
3.1 The Language To Be Edited.	3-1
3.2 Design Overview.	3-9
3.2.1 The editing commands.	3-11
3.2.2 The internals	3-12
3.2.3 Responsibilities of the constraints	3-13
4. A SIMPLE DESIGN	4-1
4.1 The Generic Constraint Types	4-2
4.2 Type Checking in Expressions and Assignment Statements	4-4
4.3 Syntax Checking.	4-18
4.4 Assignability Checking	4-22

CONTENTS (continued)

	Page
4.5 Name Binding	4-26
4.6 Extended Types	4-49
4.7 Error Handling	4-54
4.8 Summary	4-56
5. THE IMPROVED DESIGN	5-1
5.1 Removal of Environment Dependencies	5-3
5.2 Procedures Do Their Own Name Binding	5-8
5.3 Explicit Bindings	5-9
5.4 Problems	5-13
5.5 Solutions	5-19
5.6 Summary	5-31
6. SUMMARY AND CONCLUSIONS	6-1
6.1 Extensions	6-3
6.2 Applications and Future Research	6-4
APPENDIX	A-1
BIBLIOGRAPHY	B-1

FIGURES

	Page
1.1 Internal representation of a program.	1-12
1.2 An example of tree editing.	1-13
2.1 A plus constraint	2-4
2.2 A constraint network for converting temperatures.	2-5
2.3 Propagation in a network.	2-8
2.4 Propagation in a network (continued).	2-9
2.5 The underlying structure of a constraint network.	2-12
2.6 Example of a "disconnect"	2-20
2.7 Another example of a "disconnect"	2-22
2.8 Example of a "dissolve"	2-24
2.9 A type checking plus constraint	2-27
2.10 A syntax checking while-loop constraint	2-28
3.1 The search environments (one)	3-5
3.2 The search environments (two)	3-6
3.3 The search environments (three)	3-6
3.4 The program editor design	3-9
4.1 The initial state of the network.	4-11
4.2 An intermediate state of the network.	4-13
4.3 An intermediate state of the network.	4-15
4.4 The final state of the network.	4-16
4.5 A WHILE constraint.	4-20
4.6 A syntactic network	4-22
4.7 Internal representation of "A [5] [10]"	4-25
4.8 Representation of the declaration "VAR A: REAL"	4-34
4.9 An environment network.	4-40
4.10 New representation of the declaration "VAR A: REAL"	4-45
4.11 Representation of the declaration "VAR A: ARRAY [1..10] OF ARRAY [1..20] OF INTEGER".	4-50
5.1 The internal representation of a list of declarations.	5-6
5.2 The internal representation of a list of statements.	5-8
5.3 Three nested procedures	5-17
5.4 Name binding in the new design.	5-26
5.5 The new PROCEDURE constraint.	5-27

1. INTRODUCTION

1.1 A Short Story

Programming is fraught with many pitfalls, especially for (but not restricted to) novices. Indeed, it would seem that at times, our computers are doing their utmost to hinder our most valiant efforts. Take for instance the case of Joe Catastrophe ...

Joe Catastrophe was working late one night on a big programming assignment. He was tired and making many mistakes. His typing was not good even at the best of times, and tonight it seemed particularly atrocious. When he tried to type BEGIN it came out as BEIGN; FUNCTION appeared as FUMTION; WHILE, as WEHILE; and END, as ENID.

Joe was writing his PASCAL program using his system's general purpose text editor; one that keeps track of a "current line". To move up and down through his file, Joe used commands like UP10 and DOWN5 and to see a fragment of his file, he used a command like LIST15. To insert a line of text Joe had to know whether it would be inserted BEFORE or AFTER the current line. (Joe always seemed to get it wrong, and if this weren't bad enough he would often forget to leave insert mode before returning to editing. Frequently, he ended up with edit commands mixed in with his PASCAL program.)

After several hours of frustration, Joe decided his program was ready to be submitted to the compiler. However, Joe neglected to leave the editor before typing in the compile command. "Unrecognized edit command" flashed on the screen. After

recognizing his mistake. Joe left the editor, retyped the compile command and waited with bated breath.

Joe's heart sank when he saw the number of errors he had made in his program; missing semi-colons, mismatched brackets, missing ENDS, unrecognizable symbols, undeclared variables, incompatible types, too many parameters in routine calls, too few parameter in routine calls, and finally an unexpected end of file.

For the next two days, Joe was editing and compiling over and over until, finally, he got a clean compile. When he ran his program, the computer responded with "**** Error 5B22HE7. Aborted at instruction AA9C2 with contents FC92A41B". A visit to the local consultant revealed that this meant Joe had a subscript out of range somewhere. The consultant did not know exactly where, and as Joe's indentation scheme was somewhat non-standard, the consultant was reluctant to read Joe's program.

To make a long story short, Joe eventually did get his program running and handed in. Joe graduated and lived miserably ever after as a highly paid software engineer. [Anon]

Although this is not typical of a large percentage of neophyte endeavours, we have all, at one time or another, shared Joe's confusion and frustration. We all suffer from inadequate programming environments.

1.2 Inadequate Programming Environments

"Computer programming today is in a state of crisis (or, more optimistically, a state of creative ferment)" [Winograd 79]

In most software shops today, the programming environment consists of a so-called high-level language and a few simple tools: an interactive text editor, a compiler, a linker and a loader, and perhaps an interactive debugger.(1) To develop a working system, the programmer must first express his algorithm as a program in his programming language, enter the program using the editor, create a machine readable form using the compiler, fetch all the necessary run-time support routines and other unsatisfied references in his program with the linker, then finally load it into the computer's store and test it out. At each of these steps, he may encounter problems and errors which will force him to retreat to earlier steps; this leads to the cyclic repetition of many of the steps.

This process can be very confusing for the novice and very tedious for the expert. All of the tools are largely independent of each other. They often have different user interfaces and different internal representations. The communication and cooperation between the various tools is minimal. For instance, if editing, one can not get a list of one's files, if executing, one can not look at one's source code, etc. Some execute interactively (editor, debugger).

(1) In this thesis, we use the term programming environment to mean those tools associated with the actual entry and running of a program. We do not consider those tools related to project management (automatic documentation aids, library facilities, maintenance logs, etc.).

Others execute in essentially a batch mode where the user is entirely ignored (compiler, linker, and loader). Errors which could be corrected at the time of detection by the user are merely accumulated and dumped on him at the termination of the job.

The only tool that understands the language in which the user writes his program is the compiler and it will not talk to us! The other tools have been designed to be language independent, and sadly, language ignorant too.

What makes this process critical is that most programming being done today, is not the development of new programs, but rather the integration and modification of old ones. Most software dollars are being spent on the maintenance of already existing systems. Who is doing the maintenance? Most often, it is "not the original designers, but new programmers with an incomplete or inaccurate understanding of the system". [Winograd 79]

Do the tools of his trade help him to understand and modify the existing system? Short of providing him with source listings which he will be forced to stare at for hours, they offer no aid whatsoever.

In the past, this problem has been attacked by enforcing structure and modularity in one's programs. The emphasis has been on "restriction and discipline" in one's design. But at

best, this merely makes the task of staring at source listings somewhat easier. It has not made a substantial dent in what researchers refer to as the "complexity barrier". Terry Winograd's conclusion is that "ultimately the solution lies not in greater discipline but in more adequate tools". [Winograd 79]

1.3 Related Research

1.3.1 Teletype text editors

These are the lowest level of on-line editors, and until the advent of inexpensive screen terminals, were the most common. Being general purpose editors, they have no knowledge of whether the text being entered and edited represents a program, a document, poetry, or a letter to Dear Abby. So rather than provide powerful editing operations which might be useful in each of these specific contexts, the designers provided weak editing operations useful in all contexts.

Like all text editors, they provided the user with means of inserting, deleting, moving, substituting and searching for characters, character strings and lines of text. The user's interface to the editor is a command language. The user issues a command to the editor (e.g. "substitute 'good-bye' for 'hello' in line 20), which then carries out the appropriate action, "unseen" by the user. To verify the correctness of the edits, one is usually forced to issue another command to the editor to display the affected line or lines. This was generally so, because early terminals operated at very low bandwidths.

"Editing on a printing terminal requires a mental skill like that of blindfolded chess; the user must keep a mental image of the text he is editing, which he cannot easily see, and calculate how each of his editing command 'moves' changes it". [Stallman 80]

In general these editors have no ability to revoke, cancel or "undo" a command once it has been issued. A typing error in a command line usually forces the typist to reenter the line (very frustrating!).

"Frequently, a long period of acclimatization must elapse before an operator is sufficiently expert with the system to feel truly comfortable with it. In the interim, the user's feelings are akin to those associated with walking a tightrope while wearing a blindfold. Because of the often obscure nature of the interface that he is forced to employ, the operator cannot fully anticipate the consequences of the actions he performs. This leads to feelings of tension and uncertainty. Moreover, the user develops a fear of committing an unrecoverable error, and thereby becomes overly timid and cautious in his dealings with the system". [Good 81]

Not only does this lack of error prevention and correction force the user to spend most of his time and energy "fighting the system, which at times seems bent on frustrating his best efforts" [Teitleman, from Yonkel], but this sloppy, uncooperative and slow environment seriously hinders the kinds of problems a programmer will want to address.

1.3.2 Teletype program editors

"There are times when understanding can come only through the suppression of detail".
[Winograd 79]

Program (or structure-oriented, or syntax-directed) editors are a special class of editors that have specific knowledge about the syntax and semantics of a particular programming language. Like teletype text editors, teletype program editors suffer many of the same problems. Editing actions are performed "unseen" in response to commands issued by the user. Verification of edits must be done by having the affected area(s) of code specifically displayed, or alternatively the user must keep a mental image of the state of his program while editing.

However, since the system has knowledge of the language in which the user is attempting to write his program, it has many advantages over normal text editors.

It can better present the information being edited. For instance, it can do automatic indentation of the program, a task previously left to the user. It can suppress excessive detail by only displaying the upper levels of the program's structure, thus providing the user with a better overall "view" of his program. It can have better error detection

and correction facilities. Since the system knows the syntax of the language, it can prevent illegal constructs from ever being entered; since it knows the semantics of the language, it can inform the user of compile-time semantic errors (static semantics) (e.g. undeclared identifiers, mismatched types in expressions and assignment statements, mismatch in the number and type of parameters in routine calls, etc.).

These error detection tasks were traditionally left to the compiler. The integration of a program editor with the some of the capabilities of a compiler helps to eliminate the iterative process of program development discussed earlier.

With a teletype program editor, the user enters his program in much the same way as he would in a standard editor; he merely types it in. But as each fragment is entered, it is parsed and checked for syntactic errors. The user can be informed of any such errors immediately.

With a program editor, the user edits his program not in terms of textual items (characters, lines etc.), but rather with the constructs of his programming language, e.g. procedures, functions, statements, expressions, etc. It is these objects which are now the operands of insert, delete, move, substitute, and search commands.

"The editor is oriented toward structures, and does not allow the user to put in non-well-formed structures accidentally, although partial structures can be added. It provides a large variety of commands for moving around the structure, both incrementally and by search. Searches can be specified by general structural patterns and/or by character oriented specifications of lexical units. Various forms of on-line output allow the user to track easily where he is in the structure, even when interacting through a slow output device such as a teletype. Structured units can be embedded in and extracted from larger units. Iteration of operation sequences, and conditional operations are all easily expressible. New editing operations are definable as macros in terms of older ones" [Bobrow 72](1)

Several such teletype editors have been designed. In 1975, Martin D. Yonke developed a system which, when presented with a description of a language (a B.N.F.-like grammar) would produce a program editor for that language. However, the editors produced had limited semantic checking abilities (no type checking) and ran interpretively, hence were very slow. POPART is another such system, which when given a description of a language, produces not only an editor, but a parser, a lexical analyzer, a pattern matcher, a pretty printer and a file package as well! [Wile 1980] Chris Fraser and L. L. Lopez designed an interesting system for editing arbitrary combinations of many kinds of data structures (not just programs) such as vectors, binary

(1) Here, Bobrow is referring to the BBN-LISP editor, but his remarks apply equally well to all program editors.

trees, linked lists, records, character strings, symbol tables etc. Their system has been used to implement a LISP editor, a SNOBOL4 debugger, and a simple graphics editor. [Fraser & Lopez 81]

Other systems include INTERLISP [Teitleman 78], and MENTOR [Donzeau-Gouge et al. 80].

We will employ MENTOR as an example of how a teletype program editor might function.

In the MENTOR system (as in most program editors), the program being edited is not stored as lines of text, but rather as an abstract syntax tree. The program editor is in reality a tree-editor. Linear searches in a textual representation of a program correspond to an inorder traversal of the abstract syntax tree. Moves, insertions and deletions of character strings correspond (loosely) to moves, insertions and deletions of sub-trees in the abstract syntax tree.

Consider the following PASCAL program segment:

```

IF X > 0
THEN
  P (X, A [Y, Z])
ELSE
  BEGIN
    Y := Y * 2;
    X := 0
  END

```

Internally this could be stored as the tree:

Figure 1.1 Internal representation of a program.

Now suppose, the user wished to interchange the THEN statement with the ELSE statements. In MENTOR, this would be accomplished with the following command: "S2 X S3", which when translated into English says "Interchange ("X") the subtree rooted at the second son ("S2") with the subtree rooted at the third son ("S3") of this abstract syntax tree. Internally, the change would result in a new abstract syntax tree.

Figure 1.2 An example of tree editing.

If we wished to see whether our edit produced the desired effect, we could have the internal representation translated to a PASCAL textual version (this process is referred to as "unparsing" or "pretty-printing"), by typing "P", which would produce:

```

IF X > 0
THEN
  BEGIN
    Y := Y * 2;
    X := 0
  END
ELSE
  P (X, A (Y, Z))

```

or if we wanted a summary listing, we could type "P3" which would only print the top 3 levels of the abstract syntax tree:

```

IF X > 0
THEN
  BEGIN
    #; #
  END
ELSE
  P (X, A [Y, Z])

```

Similarly "P2" would result in:

```

IF X > 0
THEN
  ...
ELSE
  #

```

The command "S2 S1 I S3" says "Insert the subtree rooted at the third son of the root of the tree AFTER the first son of the second son of the root of the tree. Typing "P" would produce:

```

IF X > 0
THEN
  BEGIN
    Y := Y * 2;
    P (X, A [Y, Z]);
    X := 0
  END
ELSE
  P (X, A [Y, Z]).

```

"S1 X S2" would produce an error message because the system knows that the conditional expression of an IF statement (the first son) cannot be a list of statements, let alone a single statement (the second son). In this way, MENTOR ensures the integrity and correctness of the internal representation. We can never produce syntactically incorrect programs because MENTOR simply will not let us.

A disadvantage the reader might have perceived is the requirement to keep a mental image of the abstract syntax tree in one's head at all times and to anticipate how the editing commands will alter its structure. Although the developers of MENTOR claim that "it seems that on average a PASCAL programmer needs about a week to get accustomed to this new world of trees" and that "past this period, few return to the standard tools". We claim this is a serious drawback, which with current technology is avoidable.

1.3.3 Screen text editors

With the recent availability of fast (high bandwidth) and inexpensive video terminals, the speed and ease with which we can build and edit text files has taken a quantum leap forward. Unfortunately, some "dark age" installations use teletype text editors on screen terminals resulting in editors that, although run slightly faster than before, are still clumsy to use.

A common feature of most screen editors is a cursor which can be moved to any character location on the screen either by using arrow keys or on some systems by using a joy-stick or a mouse (more about this later!). For instance, the deletion of a character may be accomplished simply by moving

the cursor to that character and hitting the rub-out key, at which point the character disappears from the screen. In teletype editors, we would have to type a command which would specify which line the character occurred on, which occurrence of that character on that line we wished to delete, and finally what the character was. To make sure we deleted the right one, we would have to redisplay that line. (The probability of error is very high!)

On a screen editor, we KNOW we deleted the right character because we can SEE that we got the right one (instantaneous feedback). Similarly, text can be inserted where the cursor is just by typing it.

"... simple commands are acknowledged by displaying their effect. This helps the user feel the editor as an extension of his own fingers, and feel that he is touching the text directly rather than through an intermediary." [Stallman 79]

Such editors are now widely available and are a pleasure to use.

"Given such a tool, creative expression becomes a joy. Freedom of expression is improved; the ideas flow freely from mind to screen. A phenomenon of egoless composition results; the author does not have to measure the content of a passage against the agony involved in its creation. As a result, the author is more willing to modify and polish the document, even experimenting with major revisions in order to present the topic in its best light. What is truly exciting is that the technology for such editing systems is now of modest cost; one need only break

with the past way of thinking about editors." [Allen 79]

Among the more interesting screen editors are ETUDE [Good 81], Z [Wood 81], EMACS [Stallman 79] and SMALLTALK [Tesler 81].

Michael Good's paper "Etude and the Folklore of User Interface Design" offers many good suggestions for desirable features to include, and important points to consider in designing an interactive editor. Among these are the avoidance of modes ("Am I in insert mode, or am I in command mode, or am I even inside the editor?"), flexible English like command structures allowing abbreviations (easier to remember than obscure mnemonics), error prevention and correction (require confirmation from user before acting on "dangerous commands", allow user the ability to edit his command lines before they are issued, ability to cancel and "UNDO" commands), provision for on-line help, immediate feedback, etc.

"Z" is a screen editor, which while providing many of the desirable features of good screen editors, also attempts to provide some of the features of program editors. These include automatic indentation, ability to select complete syntactic units with single keystrokes, and the ability to "zoom" in and out of the program structure (corresponds to moving down and up the abstract syntax tree). By simply

knowing the keywords in the language and the amount of indentation in the source file. it can determine where a language structure begins and ends. Since it does not attempt to maintain an internal representation of the user's program its execution speed is comparable to that of normal screen text editors. However, we believe their claim that it is a 95% program editor is unfounded since it does no syntactic or semantic checking of the program.

EMACS is a similar system, which also claims to be user extensible and self documenting. While providing good facilities for text and program manipulation for many different kinds of text files, its commands for cursor motion and edits are cryptic. Although it would appear that a great deal of time is required to learn these commands, users claim that once learnt, they are easy to remember, and allow the user to specify many editing commands with very few keystrokes.

SMALLTALK-80 is not actually an editor but rather a complete integrated object-oriented software system. However, the environment provided for programming and editing is excellent and well worthy of note. The user interface is a large high-resolution bit-map graphics display which employs a technique referred to as "overlapping windows" which in effect extend the available screen space. The appearance of these windows or displays

on the screen is very much like notices stacked on a bulletin board. These windows may be moved to any portion of the screen, be moved "out" so that they appear to lie on top of the other windows or moved "back" so that they appear to be occluded by other windows. All of the information associated with a window may not actually fit in the area specified, in which case the missing information can be accessed by scrolling the window.

"The advantages of the overlapping-window paradigm are:

- o The displays associated with several user tasks can be viewed simultaneously.
 - o Switching between tasks is done with the press of a button.
 - o No information is lost switching between tasks.
 - o Screen space is used economically."
- [Tesler 81]

Editing is done with the help of a "mouse", which is a hand held device that causes a cursor on the screen to "track" the movements of the mouse on the table, enabling the user to point at objects on the screen. A passage of text can be "selected" for editing by moving the cursor to one end of the passage.

"The selection button (on the mouse) is pressed and held down while the cursor is moved to the other end of the passage. This operation is called 'draw-through', though it is not necessary to traverse intermediate characters en route to the destination. When

the cursor reaches the other end of the passage, the button is released. The selected passage is then shown in inverse video" [Tesler 81].

The user can now choose to copy, move, delete or replace the selected passage. This is done by choosing one of the available commands from the edit menu which is in another window on the screen. The choice, again, is made by moving the cursor to the desired command.

In this way, a passage of text can be selected without having to type anything at the keyboard. It is all done by rolling around a mouse and clicking buttons on its back. The editing menu on the screen eliminates the need for special function keys, or obscure key sequences required by other screen editors. The probability of error is almost zero; the visual feedback is excellent and the effort involved is minimal. (The LISP_MACHINE, PERQ and the DOLPHIN machines all have these properties).

1.3.4 Screen program editors

These editors attempt to combine the best features of teletype program editors and screen text editors. Generally, the approach taken is to store an abstract syntax tree representation of the program internally and to display

a pretty-printed version of the program on the screen. The pretty-printed version is of course produced by unparsing the internal syntax tree. Any edit command which modifies the structure of the tree also causes the display to be reconstructed. And whereas the cursor in a text editor traverses over characters and lines, the cursor movements in a program editor reflect a traversal of the internal tree. For instance, in one system called PASES [Shapiro et al. 80], the following interpretation of tree traversal commands has been bestowed on the cursor arrow keys:

- Up Move to the previous brother of this node.
- Down Move to the following brother of this node.
- Left Move to the father of this node.
- Right Move to the first son of this node.

When the user hits the "Up" arrow key, he sees the cursor move upwards on his screen. The fact that an internal "cursor" has moved to a previous brother in the underlying structure is transparent to the user. He is not forced to keep a mental image of this tree in his head. So the programmer is actually constructing and manipulating a syntax tree without necessarily being aware of it.

A feature common to most of these editors is the use of templates.

"A template is a formatted syntactic skeleton that contains the keywords, matched parentheses and other punctuation marks of the given statement (or expression) form. The template includes 'placeholders' at each position where additional code is required to complete the statement. The placeholders serve as prompts indicating the syntactic class of each component required to complete the statement.

Programs are created top-down by inserting new statements and expressions within the skeleton of previously entered templates. Syntax error detection is immediate because placeholders can only be replaced by syntactically appropriate insertions.

The text of a template is never typed; rather, it is inserted into the program by command.

Because a template is immutable, errors are not only prevented on entry, but can not be introduced after the fact. The entire statement can be moved or deleted as a unit and its constituent parts may be modified, but the concrete syntax of the statement form is unalterable." [Teitelbaum 79]

It is important to note the significance of templates. Never again will a programmer have to worry about where the semi-colons go, (and for that matter, neither will the language designer). The number of keystrokes required to enter a program can be drastically reduced, and the chance of typing errors along with it. The system can now worry about the proper bracketing of expressions and about which ELSE belongs with which IF (the dangling else problem). This is because with the template construction approach, it is clear what the programmer means. The user is freed from the

syntactic constraints imposed by the language and can therefore focus his attention on semantic concerns.

Several such systems have been developed and are now functioning. The Cornell Program Synthesizer [Teitelbaum 79] features integrated facilities to edit, file, execute and debug PL/CS programs (a highly disciplined subset of PL/1).

Errors in user-text are detected immediately. All erroneous code in the program is highlighted on the display until corrected; e.g. if a variable declaration is deleted, all usages of this variable will be highlighted on the display. PASES: a Programming Environment for PASCAL [Shapiro et al. 80], PATHCAL [Wilander 79], ALBE/P [Lewis 81], and IPE (an Integrated Programming Environment) [Feiler and Madina-Mora 81] all offer similar features.

Quite recently, [Demers, Reps, and Teitelbaum 83] have been investigating the use of "Attribute Grammars" to enforce both the syntactic and semantic rules of the language in their Program Synthesizer. They claim that such an approach lends itself well to the design of systems capable of generating editors with context dependent facilities.

1.3.5 Knowledge based editors

It would be lax not to mention some of the ideas being pursued by Artificial Intelligence people in the quest for the development of ultra-knowledgeable editors. Such research is not expected to have an immediate impact on editor designs but is expected to play an increasingly important role in the more distant future.

At the lowest level of editors, we saw editing functions such as the insertion and deletion of single or multiple characters. At the next level, in the program editors, we had editing functions that manipulated entire syntactic units. To accomplish this, the editors required an internal high-level representation of the syntactic structure of the program; i.e., the syntax tree.

Going up yet another level in sophistication, we encounter systems capable of performing even more exotic manipulations on programs. These systems are not necessarily editors in the traditional sense, but they possess abilities that would certainly be useful in an editing context.

For instance, systems exist which allow a user to improve already existing programs through the use of "source to source transformations". A transformational system attempts

to find and recognize common patterns in a user's program (programming cliches). It then applies transformations (ones that hopefully preserve the semantics of the program) to these patterns to produce an implementation which will be more efficient in both time and space (eg. transforming tail-recursive routines into iterative ones). [Balzer 81], [Cheatham 81] and [des Rivieres 80] are useful sources for more information pertaining to transformational systems.

Also there exist systems called "automatic program constructors", that actually write complete programs when given a very high level specification of the algorithm. Such a system can decide, for instance, the suitability of different ~~abstract~~ abstract data types for different contexts. See [Barstow 79] and [Kant and Barstow 81] for further details.

"The Programmer's Apprentice" [Waters 82] is a system which allows a user to interactively create and modify programs. It is intended that the user does the "hard parts" of design and implementation while the system attempts to provide assistance whenever possible. In a sense, this system is a hybrid of an automatic program constructor and a transformational system. It differs from standard editors in that it includes a library of standard cliches for program fragments and abstract data types. At any time, the user can have these pieces merged into his source program. For instance, telling it to "Implement a function SQUARE_ROOT

with a parameter NUMBER using successive approximation". generates an almost completely specified LISP function. The user merely has to insert a few additional details. The system can do this because it understands the programming cliché "successive approximation". It is one of the fragments it has stored in its library.

Program editors stored the user's program internally as a syntax tree. Here, more information is required. The Programmer's Apprentice stores the user's program (and the library fragments) as "Plans". Plans differ from syntax trees in that they describe both the control flow and the data flow of the program.

1.4 Wouldn't It Be Nice?

"... a system should help a programmer to stand on the shoulders of his predecessors, not his toes." [Bobrow 72]

So, rather than just having a programming language and a few tools, we would like the language and the tools integrated together to create a comfortable interactive environment for the user. Each tool should know what the other tools' goals are and attempt to cooperate in achieving these goals. Such an environment is referred to as a

programming system.

One never has to "leave" one tool to invoke another one. They are all immediately accessible. The tools are intelligent; they know that you are attempting to write a program in some specific language, hence can provide valuable aid in the process. For instance, when errors or problems are detected, the system can either prompt the user for a correction, suggest a possible fix, or help the user track down the cause of the error, or indeed the system may resolve the problem by itself.

Some of the tools are invoked automatically at appropriate times to perform various housekeeping chores which are traditionally left to the user. Such activity is transparent to the user; this frees him to devote more of his energy to his problem.

All of the tools communicate with the user in terms of the constructs of his programming language, not in terms of machine dependencies such as memory locations and hexadecimal values, etc.

This is accomplished by interfacing each tool to one common high level representation of the user's program. The representation generally used is the abstract syntax tree (a compact tree representation of the program). Such an organization leads to a very rich and ever expanding

programming environment. New tools can be added into the system as they are developed. The following is just a short list of the possibilities:

- o Parser
- o Semantic checker
- o Code generators
- o Syntax-directed editor
- o Interpreter/debugger
- o Pretty printers
- o Optimizers
- o Semantic preserving transformers
- o Program verifiers
- o Test generators
- o Translators to other source languages

While the parser and (to some extent) the editor work from the source and convert to the tree representation, the other tools work directly on the tree.

1.5 Thesis Overview

In this thesis, we wish to investigate the design of one tool to be used in a programming environment: a program editor. This will be an interactive editor with knowledge of the syntax and semantics of the user's programming language. It allows the user to build his program using any approach

he chooses, top-down, bottom-up, from side to side; it does not enforce any one strategy. However, it strictly enforces the structural correctness of the user's program fragments at all times and reports structural (syntactic) and static semantic errors at the time of their creation.

Although such a system is not new, we have taken a novel approach to the problem. Rather than use the standard abstract syntax tree structure as a static internal representation to be manipulated by separate processes and subsystems, we intend to represent the syntax tree as an active "constraint network" [Sussman and Steele 79] with capabilities for handling and controlling a large part of the syntax and semantic checking functions on its own.

Chapter 2 is a brief review of "constraint networks" and why they appear to be applicable to the task of high-level editing.

In chapter 3 we give a design overview of our editor. Included is a description of a small PASCAL-like language around which the editor is designed and a discussion of the roles that constraints are to play within the editor framework.

Chapter 4 presents a "first try" at a simple design and considers its advantages and disadvantages.

Chapter 5 improves upon what was accomplished in chapter 4. The design is more complicated, but results in a much more efficient system which can do more useful work on smaller fragments of programs.

We wrap up the thesis in chapter 6, by answering the philosophical questions, "Is it a good idea?", and "Could such an approach prove useful in other tools?".

2. CONSTRAINTS

2.1 Another Short Story

The following is the story of Joe Tuba's adventure in Constraint Land.

When Joe Tuba opened his eyes, he could scarcely believe what he saw. Somehow, he had ended up in a long dark tubular shaped corridor. "Where am I?", he thought.

As Joe was gathering his wits about him, he was interrupted by the sound of footsteps approaching from behind. Joe reeled around. Coming through the gloom, was a short man with a big bushy beard wearing a cap with a big number "5" on top. "What are you doing in my tube?", rumbled the man in a deep voice that echoed up and down the tube. "I'm sorry", said Joe, "I don't really know how I got here. I think I'm lost". "Hmm, most unusual. Well, my name's 5 and I'm a cell value. Why don't you come down to the depot with me and maybe the rules can figure out what to do with you". "Thanks, my name's Joe and I'm a student", said Joe, wondering what a "cell value" was and who these Rules might be. "Joe, eh?", said the man, "Nobody around these parts with a name like that. You're not a bug, are you?" Joe was too confused to be insulted. "I don't think I am".

So the two of them walked along the tube, the bearded man leading, Joe following behind (it's rather difficult to walk side by side in a tube). Before too long, they came out into a big round room. Across the room on the right and the left were two more tubes leading off into the darkness. Above the tube on the right was the sign "ADDEND1" and above the one on the left was the sign

"ADDEND2". Turning around, Joe saw that the tube they had just come out of had the sign "Sum" above it.

In the centre of the room was a high podium with three elderly gentlemen who were sound asleep. Each was directly facing one of the tubes' entrances.

5 leaned over and whispered in Joe's ear, "Those are the rules, the one facing us is Rule 1. The others are Rule 2 and Rule 3. You stay here while I go check in." 5 walked over to Rule 1 and woke him up. Rule 1 peered down at him, decided there was nothing to do and went back to sleep.

A little while later, a man wearing a cap with a "2" on it appeared at the ADDEND2 entrance, walked over to Rule 3, and woke him up. Rule 3 looked up, noticing that both an addend and a sum were present. "Let me see ... 5 minus 2 is ah ... 3!". Rule 3 then picked up the phone and dialed. "Hello, Propagator", he said, "Send us up a 3!".

"What IS going on around here?", thought Joe.

At that moment, a trap door near the podium popped open and a man climbed out wearing a cap with a "3" on it. "3, you're to head down the ADDEND1 tube", said Rule 3. Just then, however, a man whose cap bore a "6" appeared at the ADDEND1 tube entrance. The "6" crossed over to Rule 2 who looked at him and cried, "6 does not equal 5 minus 2, we've got a conflict!". Rule 2 dialed the master who told him to retract the sum. "Guards", yelled Rule2, "Take misters 5 and 3 away and dispose of them!"

Joe leaped forward, "Why are you doing this?" Rule 1 stared at Joe, Just who might you be?" "I'm Joe Tuba", answered Joe. "What are you doing here? You're not a bug are you?", said Rule 1. "Why do people keep asking me that?!", shouted Joe. "Look, I was minding my own business, when all of a sudden I ended up in that tube over there". "A likely story. Guards, take the bug away

and behead him!!"

Just as Joe was about to have his head removed, he woke up. "I wish I could stop having these nightmares about constraint networks!" [Anon]

2.2 Introduction To Constraints

"A model for computation combining a simple declarative semantics with a vivid intuitive visualization as a network of simultaneously active physical devices computing in parallel without prior prejudice as to the direction of the flow of data using the technique of local propagation augmented by dependency-directed backtracking for detecting and resolving global inconsistencies." [Steele 80]

A constraint is a declarative statement that enforces a relationship among several entities. For instance, "the combined heights of the brothers Biff and Scooter is the same as the height of their father Billy". The equation "Biff + Scooter = Billy" will instantly pop into the heads of even the most amateur mathematicians. Often very complex situations or systems can be modelled using similar equations. The model describes how the relationships between the various components of the system are constrained.

It is important to notice that a constraint equation is

NOT an algorithm for determining any of the values in the equation. The constraint equation "Biff + Scooter = Billy" enforces the same constraints on the entities as does the equation "Scooter = Billy - Biff". Nevertheless, it is true that Billy's height can be computed given the heights of his two sons; similarly, any of their heights can be uniquely determined given the heights of the other two.

Pictorially, we might represent the constraint on the heights of the three gentlemen as a box with three "pins" or "wires" attached to it as in figure 2.1.

Figure 2.1 A plus constraint.

This box constrains the sum of the values on the two pins on the left of the box to be equal to the value on the pin on the right side.(1)

So, presumably if there were values on two of the pins of this constraint, the constraint could compute a value for

(1) Notice that, as in algebra, the labels on the pins are just identifiers that presumably convey meaning to us humans.

the other pin to ensure that the declared relationship between these three entities might be maintained.

Also, one could imagine that we could combine more than one of these constraint boxes by "wiring" them together to form a "constraint network". For instance the constraint equation " $F = 32 + 9 / 5 * C$ " might be represented by the constraint network as depicted in figure 2.2.

Figure 2.2 Constraint network for converting temperatures.

Notice that the constraints are "joined" by connecting their pins. In that way, if a constraint computes a value for one of its pins and this pin is connected to pin of a neighbouring constraint, then this value becomes an input

value for that constraint. In the case of the temperature converter, given a value for "F", it would presumably calculate a value for "C" and vice-versa.

Steele and Sussman [79] have designed a system which allows a user to build such constraint networks. In the next three sections of this chapter we will examine the basic features of the system as implemented by [Steele 80].

2.2.1 Propagation

"The central idea behind constraint-based systems is the notion of 'local-propagation' - that a number of small processes arranged in a network, each processing information locally and sending results on to neighbours, can cooperatively produce a useful global effect.

The direction of computation is determined dynamically; a constraint attempts to enforce a relationship among several parameters without any prejudice as to which are inputs and which are outputs. It is willing to compute any parameter from others when those others have been determined." [Steele 80](1)

As an example of propagation, let us look at our Fahrenheit / Celcius converter more carefully. We will demonstrate how this constraint network computes $F = .68$ given that $C = 20$.

(1) Direction of computation can, however, be predetermined by designing the constraint so that computation is triggered only on certain pins.

Figure 2.3 Propagation in a network.

Since the PRODUCT and the M2 of the multiplier constraint MULT2 have values, the MULT2 constraint can compute a value for its M1 pin. Therefore M1 is assigned $1.8 (= 9 / 5)$. (See figure 2.3). Notice that the M2 of MULT1 now has this value; so the MULT1 constraint can use it to make further deductions.

Since the M1 and the M2 of the MULT1 constraint now have values, MULT1 computes a value for its PRODUCT. The PRODUCT is assigned $36 (= 1.8 * 20)$ as is depicted in figure 2.4.

Figure 2.4 Propagation in a network (continued).

The ADDER1 constraint now has values on its A1 and A2 pins and can therefore compute its SUM. The SUM is assigned 68 ($= 36 + 32$). The value of F is therefore 68.

An eager reader might now be saying, "I can see that it works, but how is it actually implemented?". "Give me more details!"

For our purposes, a brief explanation will be sufficient. An instance of a constraint is simply a data structure containing one: a list of pins which are actually links to another data structure called a cell and two: a link to the generic constraint type of this instance.

Cells are where information about the propagated values are kept. For now, let us just say that a cell contains a field to tell us if the cell does contain a value, and if so, a field to contain the actual value, and a field specifying the owner of this cell. If a cell is attached to a constraint, the entry in this field will be a reference to that constraint, and nil otherwise.

Now, whenever the owner of a cell is a constraint, and that cell acquires a value, it triggers its owner into action. The specific action that an instance of a constraint is to perform when it is activated by some cell, is recorded in the generic constraint type of that instance.

As you probably suspect, a generic constraint type is just another data structure with a name field (for instance ADDER, MULTIPLIER, etc.), a list of the names of the pins for this constraint type and a list of rules. Each rule specifies a list of pins which must have values before that rule is to be invoked (we say that the rule is 'Triggered by' those pins), and an action to perform if invoked. We might define an adder constraint in the following fashion:

```

[DEFINE_CONSTRAINT Adder
  [PINS (Sum A1 A2)]
  [RULES
    [(TRIGGERED_BY A1 A2)
     (:= Sum (+ A1 A2)))]
    [(TRIGGERED_BY Sum A2)
     (:= A1 (- Sum A2)))]
    [(TRIGGERED_BY Sum A1)
     (:= A2 (- Sum A1))]]]

```

As you can see, an Adder constraint has 3 pins Sum, A1, A2 (with cells attached to each of them) and 3 rules. If any cell owned by this constraint receives a value, a search through the rules is performed checking to see if all of the pins in each TRIGGERED_BY list have values; if so, that rule is invoked. For example if both the A1 and the A2 pins of an instance of an Adder constraint have values in their attached cells, then the cell attached to the Sum pin is assigned the plus of A1 and A2.

Now, what happens when pins from different constraints are connected? Suppose the A1 pin of some Adder constraint is to be connected to the M2 pin of some Multiplier constraint. Basically what happens is that the cell on the A1 pin and the cell on the M2 pin are bound together into a new structure called a node. If A1's cell had a value, it can now be placed into M2's cell which in turn can trigger its owner constraint (if there is one) into action. A node,

then, is simply a collection of cells which are bound together and are forced to share the same value. (Essentially a node is an equality constraint).

Figure 2.5 is the Fahrenheit / Celcius converter network redrawn, this time, showing some of the underlying structure.

Figure 2.5 The underlying structure of a constraint network.

The dashed line boxes indicate the nodes. Although in this example every node contains only two cells, Steele's implementation places no restriction on the number of cells that can be bound together in a node. Notice that, in this

example, there are cells which are not connected to any constraint. The 'F' cell and the 'C' cells are variable cells which can be initialized by the user, or whose values can be computed by the network via propagation. The cells containing 32, 9 and 5 are constant cells whose values are set by the user, but cannot be changed by the constraint network.

2.2.2 Dependencies

An extremely useful feature of Sussman and Steele's design is its ability to be accountable for its own actions. This is unlike most computer systems, where the chain of decisions and actions leading to a result are lost. The user is supplied with an answer which may or may not be correct, but if he wants to know exactly why or how the system computed the result, he is out of luck. This lack of information will be a serious impediment to future designers attempting to develop increasingly complex systems.

As we observed for both the multiplier and adder constraints, a value could be computed for any pin of either constraint given that its other two pins already had values. We say that the computed value is DEPENDENT on the other two

input values. In general, when a rule of a constraint computes the value of one of its pins, this value is DEPENDENT on exactly those pins in the TRIGGERED_BY list of that rule. The TRIGGERED_BY list may be just a subset of all the other pins of the constraint; i.e., the value of a particular pin is not necessarily dependent on the values of all of the other pins. (Note that, in the case of a node, the cells are all implicitly bound together by an equality constraint. Hence, each of the cells is dependent on the cell which first supplied the node with a value.)

Suppose the user wants to know how some cell came to have a particular value. If the user did not assign the cell a value himself, then the constraint network must have computed it. By traversing the network backwards along the paths of computation, the system can determine the origins of the value in question. For example, in our Fahrenheit / Celcius converter, if the user wanted to know how the Fahrenheit cell came to have the value 68, a constraint system could produce the following explanation:

"The value 68 is in the Fahrenheit cell because this cell is connected to the Sum cell of Adder1. Adder1 computed it using the rule $[:= \text{Sum } (+ \text{ A1 A2})]$, where $\text{A1} = 36$ and $\text{A2} = 32$.

The value 32 is in the A2 cell of Adder1 because this cell is connected to the constant cell containing 32.

The value 36 is in the A1 cell of Adder1 because this cell is connected to the

Product of Mult1. Mult1 computed it using the rule $[:= \text{Prod} (* M1 M2)]$, where $M1 = 20$ and $M2 = 1.8$.

The value 1.8 is in the M2 cell of Mult1 because this cell is connected to the M1 of Mult2. Mult2 computed it using the rule $[:= M1 (/ \text{Prod} M2)]$, where $\text{Prod} = .9$ and $M2 = 5$.

The value 5 is in the M2 cell of Mult2 because this cell is connected to the constant cell containing 5.

The value 9 is in the Product cell of Mult2 because this cell is connected to the constant cell containing 9.

The value 20 is in the M1 cell of Mult1 because this cell is connected to the Celcius cell.

The value 20 is in the Celcius cell because you put it there!

We say that 68 is ULTIMATELY dependent upon the constants 32, 9, and 5 and the Celcius value, 20. (That is, the relationship "Is_Ultimately_Dependent_On" is the transitive closure of the relationship "Is_Dependent_On".)

This process of backtracking along the paths of computation is called "dependency directed backtracking". If ever a contradiction occurs, the system can perform the same backtracking algorithm to arrive at a list of premises which are inconsistent in the context of the user's particular constraint network.

The implementation of this feature is quite straightforward. To record dependency information, the

system just has to remember where and how a cell obtained its value. A cell's value is either supplied by the user (a variable or constant cell) or deduced by a constraint. In the latter case the system must remember which rule was used to compute the value.

Where do we keep this information? In the nodes of course. It is here that all propagation takes place, so it is here that all dependency information should be recorded. We merely have to add two fields: a supplier field which contains the cell which first supplied the value to the node; and a rule field which contains a reference to the rule used to compute the value. If the owner of the supplier cell is NIL, then the cell is either a constant cell or a variable cell; otherwise, the owner is a constraint.

How do we detect contradictions? When a value is placed in a cell that is inside a node and that value is not equal to the values in the other cells of that node, a contradiction is signalled.

2.2.3 Retraction

Although the system of constraints described so far is a

powerful tool for modelling complex situations, it is the ability to retract one's assumptions, change parameters, and perform reconstructive surgery on the network that make it a truly viable tool.

"Ideally one would like to be able to invert any action with an appropriate counter-action and have the state of the system be as if the action had never taken place." [Steele 80]

Since, in this thesis, we are interested in the applicability of constraint networks in the context of a program editor, it is obviously essential that we be able to modify and tinker with the network. If we ever make mistakes, it is very useful to have a forgiving system which can "undo" them for us (perhaps automatically). Retraction appears in two forms.

In the first case, we may simply wish to place a new value in some cell and observe how other cells are affected. For instance, now that we know that 68 degrees Fahrenheit is equal to 20 degrees Celcius, we might like to know what the Fahrenheit equivalent of 40 degrees Celcius is. But alas, it is not sufficient to simply replace the 20 that is already in the C cell with 40. Since propagation has already taken place in this network, every cell already has a value. In other words, the network is "used up". In particular, the Product cell of MULT1 is 36 and the M2 cell is 1.8. Placing a 40 in the C cell (Which is also the M1 cell of Mult1)

immediately causes a contradiction to be signalled by the Mult1 constraint. (40 times 1.8 is not equal to 36.)

So before we can replace a value in some cell, we must first retract that value and ALL values dependent upon it. Once this is done, we can supply the new value and let propagation take place again.

We have already discussed the chain of events that occur when two constants are connected together. (In an editor this corresponds to an insertion). Let us now consider two procedures that break connections in the network. (Breaking a connection corresponds to clipping out a program fragment in an editor).

The first procedure is called DISCONNECT. It is used to remove one cell from a node. Structurally the only change that occurs, is inside the node. The disconnected cell is no longer bound together with the other cells in that node. Computationally, the state of the network can be changed in two ways.

One: if the cell being disconnected originally supplied the node with a value, then all of the other cells in that node are dependent upon the removed cell for their values. With that cell's removal, the values in all of the other cells must be retracted, and all cells anywhere in the network which are dependent upon those cells must in turn

have their values retracted etc. (That is, any cell which is directly or indirectly dependent upon a cell whose value has been retracted must in turn have its value retracted).

Two: if the cell being disconnected did NOT originally supply a value to this node, then this cell is dependent for its value on some other cell in that node. Hence the removed cell must have its value retracted. All cells, anywhere in the network, that are dependent upon this cell, must have their values retracted. But the values in the remaining cells of the node remain intact, because the cell that originally supplied the node with a value is still present in that node.

Perhaps a few pictures might clarify the situation! Figure 2.6 shows the before and after states of a constraint network in which the cell being disconnected originally supplied the node with a value.

DISCONNECT the Product cell of the Multiplier

Figure 2.6 Example of a "disconnect".

Notice, that because the Product cell of the multiplier constraint originally supplied the node with a value, its disconnection causes all dependent information in the node

to be retracted. Because the value in the Sum cells of both the adder constraints is dependent on these retracted values, they too must be retracted.

Notice also, that the A2 of Adder1 and the A1 cell of Adder2 are still bound together in the node even though neither have a value.

Figure 2.7 shows an example of a cell being disconnected, but this time the cell did not originally supply the node with its value.

DISCONNECT the A1 cell of Adder2

Figure 2.7 Another example of a "disconnect".

In this case only those cells dependent on the A1 cell of Adder2 were retracted.

The second procedure is called DISSOLVE and is used to

remove all of the cells from a node. It is equivalent to performing a DISCONNECT on each of the cells in a node. Upon completion, the node structure will no longer exist.

Figure 2.8 shows what happens when we dissolve the product cell of the Multiplier. (Note that the same result would transpire were we to dissolve the A2 cell of Adder1 or the A1 cell of Adder2).

DISSOLVE the Product cell of the Multiplier

Figure 2.8 An example of a "dissolve".

2.3 Constraints and Program Editors

A valid question to ask at this point (and indeed, the same question asked by the author when the topic of this thesis was first proposed) is: "What do constraint networks have to do with program editors?"

Remember, a constraint network enforces relationships between several entities. When a programmer writes a program in some programming language, he is obliged to follow the syntactic and semantic constraints of the language. If we can define a mapping between language entities and their usage rules via Steele and Sussman's constraints, then we will have the kernel of a program editor already designed for us! (Well, almost).

We propose a design of a program editor, where the internal tree representation of the user's program will in fact be a constraint network. The advantages of using this technique are two-fold. One: both syntactic and semantic errors will be detected immediately upon entry as contradictions in the constraint networks. Two: when errors do occur, the dependency analysis done by the constraint system can either resolve the problem itself or can inform

the user where the inconsistencies are located in the network (program tree).

In the constraint networks we have seen, the values being propagated were all rational numbers. Now, we have the need to propagate other kinds of "values" through the network. To ensure that the syntactic structure is maintained, we will need to propagate "syntactic class" values. To ensure that the languages data type rules are maintained, we will wish to propagate types (e.g. INTEGER, REAL, BOOLEAN) through the network. To ensure that variables are declared before they are used we will need to propagate symbol table information. While these three kinds of values are the most pervasive, one can easily envisage other kinds of information being propagated through the syntax tree: compile time constants (the constraint network could not only be used to ensure legality of usage but could actually evaluate the expressions involving these constants), values needed for address computation and values needed for code generation.

Before proceeding further, it might be useful to give an example or two of how constraints might be used to represent the nodes of a program tree. Essentially, this amounts to replacing each kind of node in the syntax tree with a constraint. For instance a WHILE node would be replaced with a WHILE constraint, a PLUS node with a PLUS constraint, etc.

In terms of type checking a PLUS constraint would be a constraint with three pins: a "Result" pin, a "Left Expression" pin, and a "Right Expression" pin (as in figure 2.9).

Figure 2.9 A type checking plus operator constraint.

The rules to be enforced by this constraint might be that the types of the left and right expressions must be either INTEGER or REAL. If at least one of them is REAL then the result type is REAL, otherwise the result type is INTEGER.

Conversely, if it is known that the result type must be INTEGER, then both the left and right expressions must be of type INTEGER. If the result type is REAL then the left and right expressions can be either INTEGER or REAL.

For syntax checking, we will need pins for the propagation of syntax information. A WHILE constraint might be a constraint with three pins: a "Parent" pin, a "Conditional Expression" pin, and a "While Loop Body" pin.

(See figure 2.10).

Figure 2.10 A syntax checking while-loop constraint.

Since a WHILE-loop is a statement, this constraint could set the value of its parent pin to be "statement", hence if this constraint were to be attached to some other constraint by its parent pin, then the other constraint could decide if "statement" was an acceptable value or not. We would expect an expression to be connected to the conditional pin, so this constraint could set the value on that pin to be "expression". Similarly, this constraint could set its Body pin to have "statement" as a value. Notice therefore, that even before any other constraint is wired to this one, we can already have values placed in all three of its cells.

3. EDITOR OVERVIEW

3.1 The Language To Be Edited

A program editor performs its editing operations on one specific programming language only. So, before designing a program editor, we need to choose a programming language.

Due to the current (misguided?) popularity of PASCAL, and because it is representative of modern "structured" languages, the toy language we have chosen is a small dialect of PASCAL. Our language is not a strict subset of PASCAL; there are some syntactic and semantic differences which we will soon discuss. (Hence the term "dialect").

The language is small enough to prevent us from being buried under an infinity of detail, yet robust enough to require us to confront and solve many of the editing problems associated with PASCAL-like languages.

As in PASCAL, the body (block) of a PROCEDURE or PROGRAM consists of declarations (variable declarations and procedure declarations) and a statement part specifying the algorithmic actions to be executed upon activation of the block. Unlike PASCAL, there is no facility for defining LABELS, TYPES, or CONSTANTS. In addition, the PROCEDURES are

parameterless and only one variable may be defined in a VAR declaration, (but, in the declaration part of any block, we allow an arbitrary number of variable declarations and procedure declarations to occur; moreover, we allow them to occur in an arbitrary order. (In a PASCAL block, all variables must be declared in a single VAR statement BEFORE any procedures are defined).

In the executable part of a block, we allow conditional statements (IF-THEN or IF-THEN-ELSE), pre-test iterative statements (WHILE loop), procedure calls and assignment statements.

Data types include the three simple types: Integer, Real and Boolean, and one structured type: Arrays, which are defined with the following syntax:

```
"ARRAY" "[" lower_bound ".." upper_bound "]" "OF"  
    component_type
```

where the lower and upper bounds must be integer constants and the component_type is either a simple type or a structured type.

In our language, the words "Integer", "Real", and "Boolean" are keywords (special symbols), unlike PASCAL where these words can be redefined in any of the user's declarations. (E.g. "VAR Real: Integer" defines a variable "Real" to be of type "Integer". Any subsequent use of the

identifier "Real" in the block in which it has been declared, will refer to this variable "Real" and NOT the type "Real".)

Expressions in our language can be constructed using integer and real constants, the boolean constants: "True" and "False".(1) variables, the arithmetic operators +, -, *, /, DIV, and MOD, the boolean operators AND, OR, and NOT, and the relational operators =, <>, <, >, <=, and >=.

Also, just as in PASCAL, every usage of an identifier in the statement part of any block must have a defining occurrence in the declaration part of the same block or in the declaration part of a containing block. We will use the phrase "Environment of an identifier" to denote those declarations which must be searched to determine that identifier's declaration. The "local environment" of a particular procedure (or program) refers to those declarations which occur in that particular procedure (or program). Unlike PASCAL, the declaration of a variable does not have to precede its usage.

To illustrate these concepts, we offer the following example. Notice that if we were to actually attempt to execute this program we would find ourselves in an infinite

(1) Like "Real", "Integer" and "Boolean", we treat "True" and "False" as keywords, hence they are not redefinable.

loop, the program is nevertheless correct in terms of syntax and compile-time semantics.

```
PROGRAM
  Outer;
    VAR a: Integer;
    PROCEDURE
      Inner1;
        VAR a: Real;
        VAR b: Integer;
        BEGIN
          a := 1.4;
          b := 2;
          c := 6.2;
        END;
    VAR c: Real;
    PROCEDURE
      Inner2;
        VAR c: Integer;
        BEGIN
          Inner1;
          Inner2;
          c := 1;
        END;
  BEGIN
    Inner2;
    a := 1;
    c := 2.0;
  END.
```

For any identifier occurring in the statement part of the procedure Inner1, the declarations to be searched would be:

Figure 3.1 The search environments (one).

The dashed lines are used to separate local environments. The first two declarations are the declarations of Inner1's local environment; these declarations will be searched first. The next four declarations are the declarations of the program's local environment (notice that Inner1's declarations are not included in the program's local environment); these declarations often referred to as the global declarations, will be searched if the declaration sought was not found in Inner1's local environment. Finally, the declarations contained inside the outer rectangle are the declarations to be searched if the search for a particular declaration is still unsuccessful. We will refer to this last local environment as the "standard environment".

identifier of the declaration, the occurrence of the variable "a" in the Inner1 procedure is defined to be of type Real, and the variable "a" in the statement part of the main program is defined to be of type Integer.

- o The name of a procedure does not appear in its own local environment; instead, it appears in the containing environment of that procedure. If it were to be defined in it's own local environment, recursive calls would be allowed, but no external procedure would have access to it, because the procedure's name would not appear in the external procedure's environment. (Were we to handle procedures with parameters, the parameter types would have to appear in the container's environment, but the parameter names would be in the local environment.)
- o Unlike PASCAL, which forbids forward references in most circumstances, the declaration "VAR C: Real" and the declaration of the procedure Inner2 are visible from the local environment of Inner1; i.e., we can refer to the variable C and the procedure Inner2 inside the body of Inner1. A PASCAL compiler would either claim that "C" and "Inner2" are undefined, or would flag these as illegal forward

references. In our language, we will allow such references. We discovered it is actually easier to allow them than to disallow them. Because of this, we have no need of PASCAL'S "FORWARD" directive.

- o As we mentioned earlier, the last environment to be searched will be the "standard environment". If we have to search the standard environment, and still cannot find a declaration for some identifier, then that identifier is undefined (a compile-time semantic error). (If we were to implement a program editor for PASCAL, we would place the declarations of all predefined constants (MAXINT), predefined types (Integer, Real, Boolean; Char) and predefined procedures and functions (Succ, Ord, Read, Write, New, etc.) in this environment.)

This process of searching for declarations will be referred to as "name binding". The search attempts to bind type information to the usage of an identifier.

3.2 Design Overview

Figure 3.4 The program editor design.

The complete design of our program editor is illustrated in figure 3.4. On the left, we have the user. His interface to the editor should be made as comfortable as possible. To this end, we envisage a cursor-oriented editor. The user directs the editor to do his bidding with a set of input devices. As we mentioned in chapter one, these might include a standard keyboard, joysticks, lightpens, mice, etc.

The user can build his program either by using the template construction method described earlier, or by entering complete fragments directly as text which will then be parsed. In either case, the result is a portion of a program syntax tree, which in our case, will be a constraint network, where the nodes of the tree are constraints, and the branches to other nodes are the wires along which propagation and retraction will take place.

A parser has been included in the design for two reasons. One, a user may already have a complete source program stored in its textual form; we do not want to force the user to re-enter his program by using templates, or any other style of construction, for that matter! The parser can produce the desired internal constraint network by reading directly from his source file. Secondly, there are situations where the template method is actually more cumbersome to use than simply entering the program fragment directly. The template method is particularly awkward for entering expressions. Typing "A [I, J] + 2 * K" requires much less thought than trying to express it as a tree, which is really what the template corresponds to.

Consider Joe Catastrophe attempting to use a "templates-only" style editor to enter the above expression.

"Now, let me see, this expression is the '*' of two things, so I'll invoke the '*' template. O.K, oops! Wait a minute, I guess it's the '+' of two things because '*'

has a higher priority than '+'. O.K., I'll get rid of that '*' and invoke the '+' template. O.K. that's done, now I have to enter the left operand of the '+'. Well, I guess that's a subscripting operation, so I'll invoke the subscripting template. O.K., now it wants me to enter the array that I'm subscripting, well that's 'A', ..."
(Five minutes later) "Gee,, what a stupid editor, why can't I just type in 'A [I, J] + 2 * K' and let it do all this work!"

Now, for output, the editor displays error messages and properly indented "pretty-printed" portions of the user's program fragments on a high-resolution display (the output device). Perhaps the various fragments could be displayed using SMALLTALK's notice-board style of presentation. With the advent of colour displays, problem areas of code could be highlighted with the use of contrasting colours.

3.2.1 The editing commands

Rather than attempt to catalogue everyone's favourite editing tricks and features here, we will consider only the absolute essentials; i.e., insertions and deletions. Substitution can be viewed as a deletion followed immediately by an insertion; a move, as a deletion (with a temporary copy made beforehand) followed by an insertion of the copy; etc.

In general, an insertion will result in two constraints being wired together, a deletion will result in two previously wired constraints being disconnected.

3.2.2 The internals

As mentioned before, internally the user's program fragments are represented as trees with constraints as the nodes. Each fragment corresponds to one network.

These networks will behave in the same way as the constraint networks of Steele. However, to implement the editor, there are a few extra details which have to be attended to, all of which cannot be handled by Steele's system as is.

The assumed goal of a session with the editor is to produce a correct, complete program. But, because we allow the user to build his program out of fragments, some of which may have missing sub-fragments (corresponding to "unfilled" placeholders in templates), and some of which may contain unresolved errors, the editor must have facilities for keeping track of all the program fragments, all of the missing sub-fragments (holes), and all of the unresolved errors. There is a correct complete program in the edit workspace if there is exactly one fragment whose root is a

program node (or program constraint), if the list of holes is empty, and the list of errors is empty. If this is not the case and the user attempts to exit from the editor, he can be warned of the consequences.

3.2.3 Responsibilities of the constraints

Perhaps the most significant aspect of our design is that the editor is not just a passive executor of the user's commands. At the core of our design is a constraint system, capable of performing both useful computations and detection of inconsistencies.

This thesis is concerned primarily with the application of constraints to the task of editing fragments of programs and at the same time ensuring that these fragments obey the syntactic and semantic rules of our language. To be more specific, we must attempt to:

- o Maintain correct syntactic structures
 - o ensure that only syntactically valid networks are built
- o Maintain correct compile-time semantics
 - o "type-checking"
 - o ensure that the operands of all operators are of

legal types.

- o ensure that the resultant type of every expression is legal with respect to its usage.
- o ensure that the resultant type of an expression in an assignment statement is assignment compatible with the type of the variable being assigned a value.
- o ensure that the identifiers being used in procedure calls are indeed names of defined procedures.
- o "assignment-reference-checking"
 - o ensure that the object being assigned a value in any assignment statement is indeed a legal variable reference (not a constant, procedure name, arithmetic or boolean expression, etc.).
- o "name-binding"
 - o ensure that all usages of identifiers in any fragment have a declaration in their environment.

Notice that we said "attempt" to ensure the adherence to these rules. Remember that the user is dealing with fragments of programs. The editor cannot ensure that the operands of all operators are of legal types if the user has not yet entered them. It cannot ensure that the resultant

type of an expression is legal with respect to its usage if, in fact, it is not being used anywhere. It cannot ensure that references to identifiers in assignment statements and procedure calls are legal if they have not yet been entered. Nor can it ensure that the usage of an identifier in any fragment has a declaration in its environment if that fragment does not include an environment; i.e., has no declarations.

Therefore, our philosophy in designing this editor is as follows: the editor will enforce the rules of the language as soon as sufficient information becomes available. Or more simply put: the editor does as much as it can, as soon as it can.

The consequence of this is that even a small program fragment such as "2 + False" will cause a type-checking error immediately upon entry, because the editor "knows" that one is not allowed to add an integer constant to a boolean constant. (To be more precise, it is the "plus-constraint" that constrains both its left and right operands to be either integer or real.) Similarly, entering "22 := A + B" would immediately cause an assignment-reference-checking error, because the "assignment-constraint" constrains the object to the left of the assignment symbol (:=) to something other than an integer constant.

4. A SIMPLE DESIGN

In the last chapter, we discussed the requirements of the constraint networks regarding syntax and semantics checking. In this chapter we will describe one design which will fulfill those requirements. First, we must define the behaviour of each of the generic constraint types to be used in our editor.

Constraints enforce relations on, and between, the values on their pins. We will see that many of the requirements of our editor can be enforced in the local context of any single constraint (no data base of global information is required).

There are, however, situations where restrictions of a global nature do need to be enforced. We must show how by wiring constraints together into networks, all of the requirements can be satisfied.

As an example, consider a plus constraint. It must ensure that the types of its left and right operands are either both integer or both real. It does not have to know whether the operands are variables or constants. It does not even have to know if they are defined or not. All of the required knowledge needed to enforce the type restrictions

of the plus operator are locally available. On the other hand, consider the usage of some identifier appearing in a deeply nested procedure, with its definition appearing in some external environment. How do we ensure the proper usage of that identifier in the local environment? If in the course of editing, its definition changes or is removed, the continued proper usage of that identifier in its local context cannot be easily checked.

The behaviour of each generic constraint type must be such that it maintains local correctness and at the same time ensures global correctness.

In the following sections we will propose an appropriate set of generic constraint types, then describe how the constraints might be used to do type checking in simple expressions, do assignability checking, syntax checking, and name binding. We will then tie up some loose ends (how to handle extended types, and how to handle error recovery), and state some conclusions before leaping into an improved design in chapter 5.

4.1 The Generic Constraint Types

For simplicity, let us assume that the user builds his

program fragments using just the template construction method. Then the simplest translation into the internal representation is to have a distinct constraint type for each template. For example, a PROCEDURE template would translate to a PROCEDURE constraint, a WHILE template to a WHILE constraint, a plus operator to a PLUS constraint, etc. This strategy works quite well, except in the case of list templates, i.e., templates which allow an arbitrary number of elements (eg. the PASCAL "BEGIN .. END" construct would have to be handled by a template which allowed the insertion of an arbitrarily long list of statements). The constraint translation of a list template would have to allow a variable number of pins to be connected to it. For now, let us ignore the problem and return to it later.

In this chapter, we will see that, in order to do type checking, some of the generic constraint types will need type pins and type rules; to do assignability checking, some of the constraint types will need assignment attribute pins; to do syntax checking, each constraint type will need syntax pins and in some cases syntax rules; and to do name binding, each constraint type will need environment pins and environment rules. So, in fact, we will be dealing with four separate networks, each forming a subgraph of the underlying abstract tree structure. Any given constraint may have pins in all four networks, e.g. a PLUS constraint

may have syntax pins, type pins, environment pins, and assignment attribute pins. We shall use the term "branch" to refer to a composite of such pins on one constraint, eg. the "Result" branch of the TIMES constraint consists of a "Result" syntax pin, a "Result" environment pin, etc. (It is the responsibility of the network constructor to match up the appropriate pins when the branch of one constraint is connected to the branch of another constraint).

In each of the following four sections where we will be discussing the four different networks, we will mention only those pins and rules pertinent to that particular network. So for instance, in the section dealing with syntax, when we say that a unary plus operator constraint has two pins, we mean two syntax pins.

4.2 Type Checking in Expressions and Assignment Statements

Type information is used in two areas of a program tree. It is used explicitly in the definitions of variables, e.g., "VAR A: REAL", and implicitly in expressions, e.g., "A + 10.5". For the moment, we will consider only the use of constraints to handle type checking in expressions.

Expression trees are composed of arithmetic, boolean, and

relational operators and operands (variables and constants). For each operator in our language, there is a corresponding generic constraint type with a set of type pins attached to it. The behaviour of an instance of one of these constraints when type values appear on their pins, is specified by a set of rules.

For instance a UNARY PLUS constraint might be defined as follows:

```
[DEFINE_CONSTRAINT Unary_Plus_Constraint
  [PINS
    (Result_Pin Expression_Pin)]
  [Type_Rules
    [(TRIGGERED_BY Expression_Pin)
      (IF Expression_Pin = Real
        THEN (SET_CELL Result_Pin Real)
        ELSE
          (IF Expression_Pin = Integer
            THEN (SET_CELL Result_Pin Integer)
            ELSE ERROR))]]
    [(TRIGGERED_BY Result_Pin)
      (IF Result_Pin = Real
        THEN (SET_CELL Expression_Pin Real)
        ELSE
          (IF Result_Pin = Integer
            THEN (SET_CELL Expression_Pin Integer)
            ELSE ERROR))]]]
```

So, a UNARY PLUS constraint has two type pins, a "Result" type pin, and an "Expression" type pin, and two type rules. The first rule is triggered if the cell on the "Expression" pin acquires a value. If the value is "REAL", then the constraint will set the value in the cell on the "Result" pin to be "REAL". If the value is "INTEGER", it sets the

"Result" pin to be "INTEGER". If it is neither (eg. "BOOLEAN") then the user has violated a type rule of our language.

These two rules enforce the rule in PASCAL that if the type of the expression of a unary plus is integer then the result type is also integer. If the type of the expression of a unary plus is real, then the result type is real.

Note that the direction of propagation in this constraint is not predetermined. It depends on which of its pins gets a value first. This behaviour is common in almost all of the constraints we will discuss. The overall behaviour of propagation in the expression network is in sharp contrast to that seen in traditional compilers. Usually the task of type checking is performed as a one pass sweep through the program tree, where type information is either passed from the top down or from the bottom up. In this system information flows in both directions. Because of this the system can do useful work even if it is only dealing with a fragment of a program. (Just given the expression "10 + FALSE", this system would detect the incorrect use of the plus operator). It is this flexibility that makes the constraint mechanism so appropriate for an editing environment.

The times, binary plus and binary minus operators will

translate to similar generic constraint types as the unary plus operator.

The real divide operator gives rise to a constraint with an unusual feature. Because the result of a real divide is always real, the corresponding constraint must ensure that the value on its "Result" type pin is always "REAL"; i.e. the value is predetermined; the value in the cell can be assigned when an instance of that constraint type is created. Its value will never need to be computed via propagation, nor should its value ever be removed by the retraction mechanism. Its value is fixed, hence it should be implemented as a constant cell. This is unlike any of the constraint types presented in Steele's work.

A REAL DIVIDE constraint might therefore be defined as

```
[DEFINE_CONSTRAINT Real_Divide_Constraint
  [PINS
    (Result_Pin Left_Expression_Pin Right_Expression_Pin)]
  [INITIALIZE
    (CONSTANT_CELL Result_Pin Real)]
  [Type_Rules
    [(TRIGGERED_BY Left_Expression_Pin)
      (IF (Left_Expression_Pin = Real) OR
        (Left_Expression_Pin = Integer)
        THEN OK
        ELSE ERROR)]]
    [(TRIGGERED_BY Right_Expression_Pin)
      (IF (Right_Expression_Pin = Real) OR
        (Right_Expression_Pin = Integer)
        THEN OK
        ELSE ERROR)]]]
```

When a constraint of this type is instantiated, the cell

on its "Result" pin will be a constant cell with its value set to be "REAL". Notice that this constraint does not propagate any information. It merely checks that the type of its expressions are acceptable, in which case nothing happens, otherwise an error is signalled. Also notice that there is no type rule associated with the "Result" pin. Since the cell on the "Result" pin is a constant cell, the "Result" pin can never be triggered, so no rule is needed.

The INTEGER DIVIDE constraint (corresponds to the DIV operator in PASCAL) is also interesting in that all of the cells attached to its pins are constant cells. Its expressions and result are always of type integer, hence there are no type rules associated with this constraint. The boolean operators are similar; their result and their expression(s) are always of type boolean.

The result of a relational operator (<, >, =, <>, <=, >=) is always boolean, but the operands can be of mixed types. For instance, we can compare real valued expressions with integer valued expressions. In fact, in PASCAL, we can even compare the ordinality of booleans since FALSE is defined to be less than TRUE. It is illegal though to compare a boolean operand with a non-boolean operand. The ASSIGNMENT constraint ensures that its right expression is "assignment compatible" with its left expression; i.e. that either the type of the left and right expressions is the same or that

the type of the left expression is real and the type of the right expression is integer. (The appendix contains the complete definitions of these constraints).

Now, what about the operands? In our language, these can be either constants or usage identifiers (representing variables or procedures). To be consistent with the rest of the design, constants are implemented internally as "CONSTANT constraints", a constraint with a single "Parent" type pin. Since there are three different types of constants in our language; i.e. INTEGER, REAL, and BOOLEAN, the value in the cell on the "Parent" type pin will be set to one of these. When the system's input handler encounters a constant, a constant constraint is instantiated, the type of the constant is determined, and the value in the cell on the parent pin is set to be that type.

Notice that the actual value of the constant that was entered by the user is not important as far as type checking is concerned. However, for pretty-printing purposes, we will store the value in an "extra" repository associated with the constraint.

Usage identifiers will also be implemented as constraints ("USAGE IDENTIFIER constraints"). Like the CONSTANT constraints, these will have one "Parent" type pin. Unfortunately, at the time of instantiation of such a

constraint we can not necessarily determine the value to be placed in the cell on this pin. This assignment will be done by the process referred to as "name-binding" which will be discussed in section 4.5.

Finally, here is an example of type propagation. Imagine the user has just entered the statement "B := FALSE OR (I * (J + 4) DIV 6 < 10 / - A)". Figure 4.1 shows the initial state of the network which will be created by the system's parser. Subsequent diagrams will demonstrate the propagation of types through the network.

The direction of propagation will be indicated with arrows. Since the constraint system sets all cells in a node to have the same value, we will only show one value, unless of course different values enter a node (causing a contradiction). For economy of space we will represent the nodes simply as dots. (The check-mark indicates that the type values meeting in a node are equal, or that a constraint was "happy" with a value it received on one of its pins).

Figure 4.1 The initial state of the network.

As you can see, the system has created a network with 14 constraints and 13 type nodes, each containing two cells. Of these 26 cells, 11 already have values. These are constant cells which received their values at the time of their creation.

Now, with values in the cells of the network, propagation will take place. Recall that if two or more cells in a node have values, the system will check to make sure that they are equal; if one of the cells in a node gets a value, all the other cells in the node which are empty are assigned that value and the owners of these cells are awakened. Depending on their rules, these constraints may simply check the new values or produce new values to be output on other pins.

The two values meeting in the node connecting the OR and LESS THAN (<) constraints are equal (both are "boolean"), so the system is satisfied and nothing further happens there. The "Result" value ("boolean") of the OR constraint is propagated to the "Right expression" pin of the ASSIGNMENT (:=) constraint. In the node connecting the OR and CONSTANT (FALSE) constraints both cells have the value "boolean", so there is no contradiction. The DIV constraint's "Result" value ("integer") is propagated to the "Left expression" pin of the LESS THAN (<) constraint, and its "Left expression" value ("integer") is propagated to the "Result" pin of the TIMES (*) constraint. In the node connecting the DIV and CONSTANT (6) constraints both cells have the value "integer", so there is no contradiction. The "Result" value ("real") of the REAL DIVIDE (/) constraint is propagated to the "Right expression" pin of the LESS THAN constraint

and the "Result" value ("real") of the CONSTANT (10.0) constraint is propagated to the "Left expression" pin of the REAL DIVIDE constraint. Figure 4.2 shows the state of the network at this point.

Figure 4.2 An intermediate state of the network.

As a result of the first phase of propagation described above, the LESS THAN, TIMES, REAL DIVIDE, and ASSIGNMENT constraints are awakened. The LESS THAN constraint is happy because its "Left expression" pin has received the value "integer" and its "Result expression" pin, the value "real". The TIMES constraint, having the value "integer" on its "Result" pin, propagates the value "integer" out on its "Left Expression" and "Right expression" pins. The REAL DIVIDE constraint is happy receiving the value "real" on its "Left expression" pin, however it does not have enough information to determine the type of its "Right expression" pin. The value must be one of "INTEGER" or "REAL", but constraints do not attempt to guess values. In this particular case, it will just wait for a type value to appear on that pin (which will be propagated up from the variable 'A'). The ASSIGNMENT constraint, upon receiving the value "boolean" on its "Right expression" pin, propagates the value "boolean" out on its "Left expression" pin.

The network is now in the state depicted in figure 4.3 below:

Figure 4.3 An intermediate state of the network.

All that remains to happen now, is for the value "integer" on the "Left expression" pin of the TIMES constraint to be propagated to the "Parent" pin of the USAGE IDENTIFIER ('I') constraint; for the value "integer" to be

2

propagated to the "Parent" pin of the USAGE IDENTIFIER ('J') constraint; and for the value "boolean" to be propagated to the "Parent" pin of the USAGE IDENTIFIER ('B') constraint. The final state of the network is shown in figure 4.4.

Figure 4.4 The final state of the network.

As you can see, during the entire propagation process, no errors were detected, hence according to the available information present at the time of propagation, the expression is consistent. However, notice that not all of the nodes contain values yet, and the process of name binding has not yet taken place. If the types of the variables 'I' or 'J' are not integer, then we will have an incorrect expression, (the same is true if the type of the variable 'A' is not integer or real).

Perhaps it should be pointed out at this time, that the process of name binding is not merely some procedure which is called at discrete points in time (as is the case in other program editors), but is implemented using the same mechanism as our type constraints. This is one of the features which distinguishes our design from all others.

The reason that name binding has not yet taken place is that the expression in our example is a stand-alone program fragment and could be inserted anywhere in a program where a boolean expression is desired. It is only when we have established a path from the definition of a variable to a usage of that variable, that name binding can occur.

It should also be pointed out, that although the expression in our example constituted a complete expression, propagation still would have taken place even in a fragment

of an expression. Our system will do as much as it can with the available information. Propagation will continue as the expression is built up.

4.3 Syntax Checking

In the previous section we saw how various operator constraints could be connected together by type pins to form a type network for expressions. Each constraint had a set of rules specifying what to do when it received type information, what type information to propagate, etc. In some cases, we saw that cells connected to these constraints could be constant cells, their values being set at the time of their instantiation. The type network was able to ensure that the expression was consistent as far as types were concerned, but now we also need a way of ensuring that the syntactic structure of the expression (and indeed the syntactic structure of any program fragment) is legal. We also have to ensure that when program fragments are connected, the resulting structure is also syntactically correct; i.e. we must be able to detect attempts to construct illegal structures.

While using the template construction method of programming prevents the user from making many syntactic

errors (spelling errors, missing semi-colons, mismatched brackets etc.), there is nothing to prevent the user from making other syntactic errors. For example, inserting an expression where a statement is expected, or conversely, inserting a statement where an expression is expected, are both illegal. "IF a := a + 1 THEN c OR b" is an example of an illegal PASCAL IF statement. A PASCAL IF statement has the form "IF <Expression> THEN <Statement>", the assignment "a := a + 1" is a statement in PASCAL, and hence is illegal in this context, and the boolean expression "c OR b" does not constitute a PASCAL statement hence is also illegal in this context.

Perhaps not too surprisingly, enforcing proper syntactic structure with the constraint mechanism is an easy task.

Remember, the syntactic structure of any PASCAL-like language is a tree structure. In our design, every constraint type will have a set of syntax pins, a distinguished pin in this set will be called the "Parent syntax pin", or the "Result syntax pin" (in the case of operator constraints). The value in the cell on this pin will indicate the syntactic class of the constraint it is attached to. All of the values in the cells on the other syntax pins of any given constraint will indicate which syntactic classes can be connected to this constraint via these pins.

Figure 4.5 shows a WHILE constraint which has three syntax pins attached to it: a "Parent" pin, an "Expression" pin, and a "Body" pin.

Figure 4.5 A WHILE constraint.

Like the DIV-constraint where all of the type values on its type pins were constants, so will be the syntax values on all of the syntax pins. The "Parent"'s pin value is "STATEMENT", which tells anybody higher up in the tree that the entire sub-tree below has the syntactic class: "STATEMENT". The value on the "Expression" pin is "EXPRESSION", which implies that for any sub-tree to be able to connect to this pin, it's syntactic class must be "EXPRESSION". The value on the "Body" pin is "STATEMENT". Any sub-tree connected to this pin must be a "STATEMENT".

Similarly, in almost all of the constraint types, the syntactic pins will have constant cells attached to them.

Building programs from program fragments is thus very much akin to playing a game of Dominoes. Either the values in the two cells which we are attempting to connect are equal or they are not. If they are, we can allow the connection to take place, if they are not, an error is signalled. In the unfortunate case that an error is to be signalled, we can provide intelligent error messages since each constraint "knows" what syntactic classes can be attached to it.

"Sorry Joe Tuba, but the body of a WHILE LOOP must be a statement. Why are you trying to declare a procedure here? We may have to behead you!"

An example of a legal syntactic network is given in figure 4.6.

Figure 4.6 A syntactic network.

4.4 Assignability Checking

The general form of an assignment statement in PASCAL is "<Expression> := <Expression>". Although syntactically, both sides of the assignment operator are expressions, semantically, they are quite different. The semantics of <Expression> := <Expression> is "put the value of the right hand expression into the location denoted by the left expression". This implies that the evaluation of the right

hand side yields a computed value and the evaluation of the left hand side yields a location in a machine's memory. (These are sometimes referred to as "r-values" (for right value) and "l-values" (for left value) [Aho, Ullman '78]).

The objective of assignability checking is to ensure that the left expression does indeed denote a location; i.e. that it is indeed a variable reference. "Why the fuss?" interrupts Joe Tuba, "Isn't the object on the left hand side of an assignment statement just an identifier, in which case all you have to do is make sure that it has been declared as a variable and not as something else like a label or a procedure?" Unfortunately, Joe, the answer is no. The object on the left hand side can be much more than just an identifier, in fact, it can be arbitrarily complex. In PASCAL, we can select a particular element of an array, we can dereference a pointer and we can select a particular field of a record structure. For example, "Date".Month [5].Day [1] := 2" is a perfectly valid assignment statement in PASCAL (subject to type compatability constraints, of course).

In our toy language, we have chosen to ignore pointers and records, but we still have array selection to worry about. In our language, "A [I + 2] [J * 6] [-K] := 5" is a legal assignment statement.

So, how do we employ the constraint mechanism to ensure that the expression on the left hand side of the assignment operator is indeed a variable reference? With an assignment attribute network of course!

This network runs parallel to the type network; i.e. all operator constraints will have a "Result" assignment attribute pin, unary operator constraints will have a single "Expression" assignment attribute pin, binary operator constraints will have "Left" and "Right" expression assignment attribute pins. CONSTANT and USAGE IDENTIFIER constraints will have a single "Parent" assignment attribute pin. The ASSIGNMENT constraint will have "Left" and "Right" expression assignment attribute pins as well.

The "Left expression" assignment attribute pin of the ASSIGNMENT constraint is a constant cell whose value is set to "TRUE" upon instantiation. The "Result" assignment attribute pins of arithmetic, boolean, and relational operator constraints are all connected to constant cells which are all set to "FALSE" upon instantiation. (We cannot assign into arithmetic, boolean, or relational expressions). Similarly the "Parent" assignment attribute pin of CONSTANT constraints is set to "FALSE". (We cannot assign into constants). The "Parent" assignment attribute pin of USAGE IDENTIFIER constraints will be set during name-binding. If the usage identifier has been declared as a variable (VAR),

it is set to "TRUE", if it has been declared as a procedure, it is set to "FALSE". (We cannot assign into procedures).

Finally, the "Parent" assignment attribute pin of the ARRAY-SELECTOR constraint is set to "TRUE". This is a constraint we have not yet considered. It has a "Parent" pin, "Left" and "Right" expression pins for type checking, syntax checking, assignability checking and name binding. The left expression denotes the particular array whose element(s) we are selecting and the right expression denotes which actual element(s) we are selecting. For example "A [5] [10]" would be represented internally as the network shown in figure 4.7.

Figure 4.7 Internal representation of "A [5] [10]".

Notice that none of these constraints has any rules concerning assignability, hence no propagation of assignability attributes ever takes place (except inside nodes). So, like syntax checking, assignability checking is

much like playing Dominoes. The "Left expression" assignment attribute of an assignment constraint is set to "TRUE". Any expression tree (network) whose "Parent " or "Result" cell has the value "TRUE" is a valid candidate for attaching to the left side of the assignment. If the value is "FALSE" it is not. Hence the following examples are valid assignments (assuming "A", "B", and "C" are declared as variables):

```
"A [I + 2] [J * 6] [-K] := 5 + J;"
"B := 5;"
"C [2] := C [3];"
```

However, the following are invalid assignment statements (assuming "P" is declared as a procedure):

```
"P := 10;"
"5 := A;"
"(TRUE AND B) := 32;"
"A [I] * A [J] := A [K];"
```

4.5 Name Binding

Name binding is the process of associating the definition of an identifier with each of its usages. The identifier might represent a variable, a constant, a procedure, a function, a label, and in some languages a type.

In most PASCAL-like languages, the declaration of an identifier consists of its classification (PROGRAM, PROCEDURE, FUNCTION, CONSTANT, VARIABLE, LABEL, TYPE, etc.),

an identifier (which we will refer to as a "definition identifier"), and a definition. In the case of a PROCEDURE declaration, the definition consists of a list of definitions local to that procedure and a body of executable statements. In the case of VARIABLE declaration, the definition consists merely of a type, eg. VAR X: REAL.

Implicit in all of these declarations is the assignability of the object being defined. According to PASCAL's semantic rules, we can only make assignments to variable identifiers. We can not make assignments to procedures, functions, constants, labels or types. (Note that this is not the case in every language).

A usage of an identifier is any occurrence of that identifier in any executable statement in the program.

In our simple language, we can define only variables, procedures and, of course, programs. So, the usage of an identifier is restricted to variables occurring in assignment statements and expressions, and procedures occurring in procedure calls.

The process of name binding is needed to ensure the proper usage of these identifiers. If "X" is declared as a variable, then we can not invoke it in a procedure call. Similarly if "Y" is declared as a procedure, then we can not reference it in any expression, nor can we attempt to assign

anything into it.

In a compiler, the process of name binding is straight-forward. Traditionally, a symbol table is generated by scanning the definitions in the program. The symbol table associates a list of attributes with each identifier in the program (its classification, its type, and its environment).

Any usage of an identifier can be determined to be correct by first performing a "look-up" of the identifier in the symbol table, then checking that the classification and the attributes associated with the identifier are appropriate to the identifier's usage. (Note that the symbol table "look-up" is organized so that only the environment in which the identifier occurs will be searched; i.e. the local environment, the containing environment, the container's containing environment, and so on, till the standard environment is reached).

In our editor, we wish to perform essentially the same operations. However we face several problems.

1. We are dealing with fragments of programs (and in particular, fragments of definitions).
2. These fragments are not static, but are subject to change at the whim of the user.
3. When fragments are joined or taken apart, name-binding

may have to be "un-done" and "re-done".

To aid our discussion of the problems we face, we present the following program.

```
PROGRAM Example;
  VAR a: REAL;
  PROCEDURE Outer;
    VAR b: INTEGER;
    PROCEDURE Middle;
      VAR c: REAL;
      PROCEDURE Inner;
        VAR d: REAL;
        BEGIN
          a := a + b + c + d
        END {Inner}
      BEGIN
        a := a + b + c
      END {Middle}
    BEGIN
      a := a + b
    END {Outer}
  BEGIN
    a := a
  END {Example}.
```

First let us consider the consequences of adding a variable declaration.

Suppose the user attempts to declare a variable "c" of type integer in the "Middle" procedure. Clearly, this must be flagged as an error, since there is already a declaration for "c" in the local environment of the procedure "Middle" (namely "VAR c: REAL").

Suppose he declares a variable "b" of type real in the "Middle" procedure. This is legal, since there is no other

occurrence of an identifier "b" declared in the local environment of "Middle". However we now have a problem: before this insertion was made, the usage of "b" in the "Inner" procedure (in $a := a + b + c + d$) and the usage of "b" in the "Middle" procedure (in $a := a + b + c$) were bound to the definition of "b" in the "Outer" procedure; i.e. before the insertion "b" was considered to be an integer. These usages must now be unbound and then rebound to the new definition of "b"; i.e. "b" in the "Inner" and "Middle" procedures will now be considered to be of type real.

Suppose he declares a variable "d" of type integer inside the "Middle" procedure. This is legal, since there is no other occurrence of an identifier "d" in this environment. Before this new declaration, the usage of "d" in the "Inner" procedure (in $a := a + b + c + d$) was bound to the definition of the "d" in the "Inner" procedure. Adding a declaration for "d" in the "Middle" procedure does not change this binding. The usage of "d" in the "Inner" procedure stays bound to the definition in the "Inner" procedure.

The general rule, of course, is that in searching for a definition for a particular identifier, we first search our current local environment, then outwards to our container's environment, then to our container's container's environment etc. The definition that we bind our identifier-usage to,

will be the first (or "closest") definition we find in our search. If in the course of editing a new definition is inserted which is "closer", we must bind the usage to the new definition; if the new definition is not closer, no change is required.

Notice also, that if a set of usages of some identifier is unbound, because currently there is no declaration for that identifier in the program fragment, then a compiler (and this editor too) would signal these as "undeclared identifiers". It would be the responsibility of the user to remove all of the usages of this identifier or to supply a declaration for the identifier.

Now, let us consider the consequences of deleting a variable declaration. (Since a declaration consists of three parts: the classification, the identifier, and the definition, any portion of the declaration is subject to being deleted).

In a case where the entire declaration is deleted or just the identifier part is deleted, the general rule still hold true. Suppose the declaration for "c" in the "Middle" procedure is deleted. All usages of "c" in the "Middle" procedure and all usages of "c" in any contained procedures (like "Inner") become unbound. These must be rebound to a declaration of "c" in an environment external to the

environment of "Middle". If there is no such declaration for "c", then all of these usages are flagged as being undeclared.

If just the classification is deleted, we are left with nonsensical fragments of programs (eg. "a: REAL" or "Inner; ... END {Inner}"). We will arrange it so that this results in the deletion of the entire declaration (as far as name binding is concerned).

Finally, consider the case where just the definition part of the declaration is deleted. For example, suppose the user deleted the "REAL" from "a"'s declaration in the "Example" program. Since the identifier in the definition has not changed, all usages of this identifier should remain bound to this declaration. However, when a new definition is supplied (in this case: a type), usages of this identifier which may have been correct as far as type and assignment compatibility are concerned, may now be in error. We conclude that all usages of an identifier must be rechecked when a new definition is installed.

Now, how can we use the constraint mechanism to achieve name-binding in a program editor environment? From the various scenarios presented in the last few pages, we see that name binding is something which must be undone and re-done whenever an environment changes. The concepts of

retraction and propagation should spring into your minds at this point! What we propose to do is propagate local environments in an environment network. This is a new network, distinct from the syntax, assignment, and type networks.

Conceptually, an environment will be propagated from the procedure's local declarations to the usage identifiers occurring in the executable part of that procedure. As mentioned before, a usage identifier is implemented internally by the system as a constraint. It now has four pins attached to it, a "Parent" syntax pin, a "Parent" type pin, a "Parent" assignment attribute pin and finally a "Parent" environment pin. When the environment arrives on its "Parent" environment pin, a rule will be triggered. Unlike all of the other rules we have previously seen, which generally do nothing more than set the value in some cell, here we have something slightly more complicated. The USAGE IDENTIFIER constraint must do a look-up in the newly arrived environment to determine if the identifier exists in the local environment. If a match is found, we can then extract the definition of the identifier; i.e. its type and its assignment attribute. The type value is placed on the constraint's "Parent" type pin and the assignment attribute is placed on the constraint's "Parent" assignment attribute pin.

If a match is not found, then a search through the container's environment is performed, and so on, until we reach the standard environment. If no match is found, we can signal an error.

The key now, is to design the propagation of environments, such that, if ever the environment changes (either by insertions or deletions of declarations or parts of declarations), then whatever environment has been propagated from the point of the deletion or insertion is retracted. To accomplish this, we must determine how we are to internally represent declarations.

Well, first, we would like to treat a variable declaration as a unit, something that can be inserted and deleted. This suggests that a declaration should be a tree. For example, a variable declaration "VAR A: REAL" might be represented internally as in figure 4.8.

Figure 4.8 Representation of the declaration "VAR A: REAL".

Secondly, we must provide a path in the environment network from the declarations of a procedure to each usage

identifier in the executable part of that procedure. The local environment will be propagated along these paths from the declarations to the usage identifiers. This means that the environment arriving at a cell of a usage identifier constraint is dependent upon the declarations. Hence, if a declaration is removed, this environment will be retracted (which is what we want) and when a new declaration is put in its place, a new environment will be propagated causing the usage identifier constraints to "look themselves up again".

Thirdly, what exactly are the values which will be propagated in the cells of the environment network? Any local environment of a procedure consists of a list (possibly empty) of declarations. And as mentioned before, each declaration consists of three parts, a classification, an identifier, and a definition which for variables consists of a type, and for procedures consists of a list of declarations and an executable body.

Unfortunately, this is a lot of information to have to propagate through a network. (The values in the cells would themselves be constraint networks!) Remember, types and procedures can be arbitrarily complex which implies that their internal network representation can be arbitrarily large. Should the environment network propagate such large data structures? To do so poses several problems. Exhaustion of the available memory supply is one. Changing

environments is another. Do not forget that we are dealing with an editing system here. A user may have a variable declared to be an "ARRAY [1..10] OF ARRAY [11..20] OF ... OF ARRAY [91..100] OF REAL" and at any time may decide to change the "REAL" into "INTEGER". If this declaration were propagated as part of a local environment, and this local environment were the current value of many cells in the environment network, we would either have to search for all occurrences of this declaration in all of these cells and change the "REAL" into "INTEGER", or we would have to retract the local environment, then repropagate another massive amount of information, or just retract that part of the environment that needs changing, then repropagate just the updated portion.

Clearly, none of these solutions is very attractive. They all deviate from Sussman and Steele's system where a cell's value never contains more than just a simple value e.g. an integer.

Well, why do we want to propagate all of this information anyway? "So the usage identifiers can look themselves up!", answers Joe Catastrophe. Yes, but rather than propagating all of the information to the usage identifiers, let us just propagate a piece of information which will tell them where to look for the answers; i.e., a pointer. The pointer will always point to the start of the list of declarations for

the local environment, which will be either a PROCEDURE constraint or a PROGRAM constraint.

Okay, now that all of that is settled we are in a position to define the constraints needed for environment propagation. At the root of the constraint network representing a program will be a program constraint. It has three environment pins, a "Parent" environment pin, a "Name" environment pin, and a "Body" environment pin. All three are connected to constant cells. The value in both the "Parent" cell and the "Name" cell is a pointer to the Standard Environment. If ever a search through the program's local environment fails, this pointer will give us access to the Standard Environment; the implication here, is that the Standard Environment is just another list of declarations, the difference is that this list must be created before the editing session starts and that the user is not allowed to modify it during the session.

The value in the "Body" cell is a pointer back to the PROGRAM constraint itself (in the appendix, this is expressed as "^ME").

A list of declarations can be attached to the "Body" branch followed by the executable part of the program (often referred to as the "main" procedure). We mentioned earlier a problem with lists. The way we propose to handle it is by

adding a "Next" branch onto the declaration constraints; i.e. the VAR constraint and the PROCEDURE constraint. To handle lists of executable statements, each statement constraint will have a "Next Statement" branch.

So, a VAR constraint has an environment cell for the "Parent", the "Name", the "Type" and the "Next" pins. When an environment value arrives in the "Parent" environment cell, this will trigger a rule which will propagate this value out to the "Name" environment cell, the "Type" environment cell and the "Next" environment cell.

The PROCEDURE constraint is similar except that instead of a "Type" environment pin, it has a "Body" environment pin. When an environment value arrives in the "Parent" environment cell (this will be a pointer to the procedure's containing environment), a rule will be triggered which will propagate the pointer to the "Name" environment cell, and to the "Next" environment cell. It will also propagate the local environment (a pointer to itself) to the "Body" cell.

All statement and expression constraints will also have environment pins wherever they have syntax pins. When they receive an environment on their "Parent" (or "Result") environment pin, they merely propagate it out on all of their other pins. Eventually the USAGE IDENTIFIER constraints will receive an environment pointer on their

"Parent" environment pins. This will trigger the rule that will make the identifier look itself up.

"I'd like to see an example.", says Joe Tuba. Well, suppose the user had entered the following program:

```
PROGRAM Catastrophe;  
  VAR A: REAL;  
  PROCEDURE Tuba;  
    VAR B: INTEGER;  
    BEGIN  
      A := B  
    END;  
  BEGIN  
    A := A + 1  
  END.
```

The internal representation showing (only) the environment network is depicted in figure 4.9. (Here, the "¹" represents a pointer to the standard environment, "²" represents a pointer to the PROGRAM constraint, "³" represents a pointer to the "PROCEDURE" constraint, and the arrows as usual, represent the direction of propagation).

Figure 4.9 An environment network.

Notice that the "Parent" environment pin of the PROCEDURE constraint is connected to the "Next" environment pin of a VAR constraint. Similarly, the "Parent" environment pin of the ASSIGNMENT constraint in the main program is connected to the "Next" pin of the "PROCEDURE" constraint. The "Next" environment pins of both the ASSIGNMENT statements are not connected to anything. (This represents the end of the "list"). (See the syntax rules in the appendix for these constraints to see how we ensure that a procedure's declarations appear BEFORE its statements).

Now, what exactly happens when a USAGE IDENTIFIER constraint receives an environment value on its "Parent" environment pin? Take for example the usage identifier "A" in the assignment statement "A := B" inside the "Tuba" procedure. It received a pointer to the PROCEDURE constraint. This is where the search will begin. Note that the search will take place in the network itself, NOT in some external symbol table area. First, we follow the "Body" pin down to the first declaration: a VAR constraint. The identifier connected to this constraint is the "B"; not the one we are looking for, so we follow the "Next" pin to the ASSIGNMENT constraint. This is not a declaration constraint, so we have run out of local declarations in this procedure, and we return to the PROCEDURE constraint.

Looking in the "Parent" environment cell we find a

pointer to the PROGRAM constraint. The search will continue from there. As before, we follow the "Body" pin down to the first declaration, again a VAR constraint. The identifier connected to this constraint is the "A". So we have found it! Since we have determined that "A" is a variable (it is connected to a VAR constraint) we can set the "Parent" assignability attribute pin of the usage identifier constraint to be "TRUE" and since the type of the VAR constraint is connected to a REAL TYPE constraint, we can set the type pin of the USAGE IDENTIFIER constraint to be "REAL".

The USAGE IDENTIFIER constraints are therefore the locations where inputs from one network (the environment network) are propagated to other networks (the type network and the assignability network). As a result USAGE IDENTIFIER constraints are also the locations where retraction of values in one network will cause retraction in other networks; i.e. values in the type and assignability networks can be dependent upon values in the environment network.

At this point we would be finished describing the four different networks, except that the design we have presented for the VAR constraint has environment bugs in it.

Suppose a user of our editor decides he does not like the declaration "VAR A: REAL", he wants to change the "A" into a

"B". So he deletes the "A", and replaces it with a "B". What has happened internally?

First, the DEFINITION IDENTIFIER constraint for the "A" is disconnected, and as we saw in chapter two, this will cause all values dependent upon the original input to the node connecting the VAR constraint to the DEFINITION IDENTIFIER constraint to be retracted. So the value in the "Parent" environment cell of the DEFINITION IDENTIFIER constraint will be retracted. Then a new DEFINITION IDENTIFIER constraint for "B" is instantiated, with its "Parent" pin connected to the "Name" pin of the VAR constraint. Propagation will cause the value in the "Name" environment cell of the VAR constraint to be copied into the "Parent" environment cell of the DEFINITION IDENTIFIER constraint. But this is all that will happen. There are now USAGE IDENTIFIER constraints implicitly bound to the variable "A", a variable which no longer exists!

The problem has arisen because the removal of the DEFINITION IDENTIFIER constraint for "A" did NOT cause retraction of the environment from the executable part of that procedure. (In particular, retraction has not taken place at the USAGE IDENTIFIER constraints).

We will encounter the same problem if a type branch is disconnected from a VAR constraint. If the user decided "A"

should be of type integer, the replacement of the REAL TYPE constraint with an INTEGER TYPE constraint would result in all usages of "A" still being bound to a real variable and not an integer one.

Why does retraction not occur? Remember, cell values are retracted if, and only if, they are directly or indirectly dependent upon a value in a cell that has been removed.

The way we have defined our VAR constraints, the value in the "Next" environment cell is not dependent upon the value in the "Name" environment cell, nor is it dependent upon the value in the "Type" environment cell. Hence disconnecting either will not effect the "Next" cell. If the "Next" environment cell is not retracted then no environment cell in the entire subtree to the right will be retracted since they are all dependent upon this one.

The solution is to force the "Next" environment cell to be dependent upon both the "Name" and the "Type" environment cells. We accomplish this by making the "Next" environment cell dependent upon the type environment cell and the type environment cell dependent upon the "Name" environment cell. This is done by adding an extra pin on the "Name" branch and an extra pin on the "Type" branch.

Figure 4.10 contains our new internal representation of a declaration such as "VAR A: REAL".

Figure 4.10 New representation of the declaration "VAR A:REAL".

When an environment value arrives at the VAR's "Parent" environment cell a rule is triggered which copies this value into the VAR's "Name" environment cell which is then copied into the DEFINITION IDENTIFIER's "Parent" environment cell. (Remember a node enforces an equality constraint on all of its cells).

This causes a rule in the DEFINITION IDENTIFIER constraint to be triggered, that first determines if this identifier has already been declared in this local environment. To accomplish this a look-up similar to that done for usage identifiers is carried out, except that here, only the local environment need be searched. If the name of

the identifier already exists, then we can issue an error. Assuming the identifier is a new one, the constraint will then place the value "A" into the "Actual_Name" cell which is then copied into the "Actual_Name" cell of the VAR constraint. This in turn triggers a rule in the VAR constraint which copies the value in the "Parent" environment cell into the "Type" environment cell.

The node forces the "Parent" environment cell of the REAL TYPE constraint to take on this value. This triggers a rule in the REAL TYPE constraint which places "REAL" into the "Type" type cell which is then copied into the "Type" type cell of the VAR constraint.

Finally, this will trigger a rule in the VAR constraint that will propagate the original environment value that arrived on the "Parent" pin, out to the "Next" environment cell and so on to all the other declarations, then onto the executable statements in this procedure.

("Boy! Is this author ever explicit!" thought Joe Tuba).

So, essentially what we have done is create a path for the environment from the VAR constraint through the DEFINITION IDENTIFIER constraint back to the VAR constraint, through the TYPE constraint, back to the VAR constraint then onto the next definitions. (Except, that the DEFINITION IDENTIFIER passed up its actual name, and the TYPE passed up

its actual type. They could have propagated the same environment value that they received, it is just that this way, the VAR constraint now has immediate access to its name and type, which will speed up usage identifier lookups).

Now, if ever a definition identifier or type is removed from a declaration of some procedure, the environment will be retracted from all subsequent declarations and statements in that procedure. When the name or type is installed, the environment will once again propagate through the network causing all usage identifiers in that procedure to look themselves up again. If ever a name or type is left out of a declaration, the propagation of the environment will be halted until that missing element is entered by the user.

Notice that a PROCEDURE constraint does not propagate an environment value (to its local declarations and executable statements) UNTIL it has received an environment on its "Parent" environment pin. This implies that a local environment is dependent upon its containing environment. (Retraction of a local environment, will cause retraction of ALL contained environments). As a result, environment propagation will only occur in those program fragments that actually contain a PROGRAM constraint, since the PROGRAM constraint does not require triggering in order to propagate its environment. So, if a user is working on a fragment that consists of just a procedure, then no name binding will

take place inside this procedure; not even if all the usage identifiers in that procedure are declared locally. Regretably, it is only when this procedure is connected to a fragment that does contain a PROGRAM constraint, will name-binding take place.

A corollary of all of this, is that any modification to a declaration in the main program will cause the environments in the entire program tree to be retracted. The same environment values will be repropagated throughout the tree when the modification has been completed.

Although this may appear to be an extremely inefficient process (and it is), we have designed it this way to ensure that any usage of an identifier will be bound to its most current declaration at ALL times. We will see how this problem can be remedied in the next chapter. (Essentially, what is required, is to somehow partition the usages of identifiers into two classes, those that are declared locally and those that are declared externally. The local ones could be looked up right away, and the external ones could wait until the external environment arrived).

4.6 Extended Types

The type values we have used so far have all been simple (boolean, integer, and real) and were all predefined in our language. In any PASCAL-like language, types can be complicated. In fact, user-defined types can be arbitrarily complex; i.e. the internal tree representation of the definition can be arbitrarily deep.

Memory restrictions prevent us from propagating such structures ~~through~~ the type network, so, as we did with environments, we shall do with types; i.e., we shall propagate pointers to the type definitions. Instead of propagating "INTEGER" as a type value, we shall propagate pointers to INTEGER TYPE constraints; similarly for reals and booleans.

In our Toy language, the only user-defined types are the array types. However, the array type is sufficient to illustrate the problem of large internal structures. Consider the internal representation of the declaration "VAR A: ARRAY [1..10] OF ARRAY [1..20] OF INTEGER":

Figure 4.11 Representation of the declaration
 "VAR A: ARRAY [1..10] OF ARRAY [1..20] OF INTEGER".

Recall from the previous section that a type constraint propagated its value (now revised to be a pointer to the type constraint) up the type network. This will be used to advantage in the more complex array constructor type.

An ARRAY CONSTRUCTOR TYPE constraint has four branches, the "Parent", the "Lower bound", the "Upper bound", and the "Element type". The rules of the ARRAY CONSTRUCTOR TYPE constraint will be such that when an environment arrives on its "Parent" environment cell, it will copy this value into its "Element type" environment cell. So if we have one or more ARRAY CONSTRUCTOR TYPE constraints connected together the environment will propagate down to the "Element type"

environment cell of the last ARRAY CONSTRUCTOR TYPE constraint. This cell must be connected, in the simplest case, to a "Parent" cell of one of the INTEGER, REAL, or BOOLEAN type constraints. These constraints, upon receiving an environment set their "Parent" type cell to contain a pointer to themselves. The ARRAY CONSTRUCTOR TYPE upon receiving one of these pointers on its "Element type" type pin will in turn place a new pointer to itself on its "Parent" type pin and thereby handle the more complex case.

By maintaining this circular path in our structure we ensure that any modification to the structure will result in retraction of the environment from all subsequent declarations and statements as desired.

So, when a USAGE IDENTIFIER constraint does a look-up for the variable "A" (say), the value that the USAGE IDENTIFIER constraint will place in its "Parent" type cell will be the value it finds in the type cell of the "Type" branch of the VAR constraint, which will be a pointer to the first ARRAY CONSTRUCTOR TYPE constraint.

We now have a problem. Suppose there is a declaration for another variable "VAR B: ARRAY [1..10] OF ARRAY [1..20] OF INTEGER", furthermore suppose the user has an assignment statement in his program such as "A := B". (Internally this will be represented by an ASSIGNMENT constraint with a USAGE

IDENTIFIER constraint connected to its "Left expression" branch for "A" and a USAGE IDENTIFIER constraint connected to its "Right expression" branch for "B"). The type rule in the ASSIGNMENT constraint insists that the type values on its left and right expression pins be equal, but clearly this is not so. The type value in the "Left expression" cell will be a pointer to "A"'s definition, the value in the "Right expression" cell will be a pointer to "B"'s definition, and yet according to the rules of our language, we should be able to assign "B" into "A" since their type definitions are the same; i.e. "ARRAY [1..10] OF ARRAY [1..20] OF INTEGER".

The solution is to modify our system's definition of equality. We say that two type values (pointers) are "EQUAL" if and only if the two structures that are pointed at are the same; i.e. exact copies of the same structure. (Those familiar with LISP will recognize this as the "EQUAL"

function, as opposed to the "EQ" function).(1)

One other problem we must address is type propagation in an ARRAY SELECTOR constraint. In an ARRAY SELECTOR constraint the "Left expression" pin is connected to an expression denoting the array which is to be selected for indexing. The value in the "Left expression" cell should therefore be a pointer to the definition of that array; i.e. it should be a pointer to an ARRAY CONSTRUCTOR TYPE constraint. (If it is not, the ARRAY SELECTOR constraint will signal an error). It is the responsibility of the ARRAY SELECTOR constraint to set the type value in its "Parent" type cell to be the type of the element of the array we are indexing. How do we determine what this is? Since the value in the "Left expression" cell is a pointer to the definition of the array we are selecting, we can find

(1)

Comparing large structures to determine their equivalence is obviously a slow procedure. Notice, that the assignment made above would be illegal in PASCAL. PASCAL would consider the two definitions to be distinct, hence NOT equal. PASCAL gets around this problem by associating a type identifier with a type definition. For instance we could define "Small_Array" to be an ARRAY [1..10] OF ARRAY [1..20] OF INTEGER" with a TYPE declaration "TYPE Small_Array = ARRAY [1..10] OF ARRAY [1..20] OF INTEGER". If both the variables "A" and "B" had been declared to be of type "Small_Array", then the type value used in the type network for "A" and "B" would both point to the same definition. Hence, a PASCAL editor using our design would define two types to be "EQUAL" if the two pointers are equal. Comparison of large type definitions for equality would not be necessary.

the element type in the definition by following the pointer to the ARRAY CONSTRUCTOR TYPE constraint and find the value in its "Element type" type cell. So, for example if "A" is defined to be an ARRAY [1..10] OF ARRAY [10..20] OF BOOLEAN then the type of A [4] is "ARRAY [10..20] OF BOOLEAN", and the type of A [4][5] is "BOOLEAN".

4.7 Error Handling

Errors can occur in six different ways in our editor:

1. A constraint attempts to place a new value into a cell which already has a value different from this one.
2. A constraint receives a value on a pin and the rule triggered by this pin determines that this value is not in the set of legal possible values.
3. Lookup of a usage identifier fails (undeclared identifier).
4. Lookup of a definition identifier succeeds (multiply defined identifier).

5. User references the program name.

6. User attempts to index a non-array variable.

Rather than have the entire editor grind to a halt whenever an error occurs, and have to wait for the user to supply a fix, we suggest that a less annoying method of handling errors is to just have the pretty printer highlight the fragment of the program containing the error. This could be done with reverse video, or increased intensity, or perhaps even different coloured letters (obviously the solution chosen is a device dependent one).

Should the user be confused by a particular error, he should be able to query the system for an explanation. ("Why has my whole program gone into reverse video?" wondered Joe Catastrophe). But because our system is based on constraints, not only can we provide the user with an explanation for the error in terms of its local context, we can have the system backtrack to find the ultimate premises of the error.

Consider the expression " $((6.2 + 5) * 1) \text{ DIV } 4$ ". It could occur (in the underlying network) that the node connecting the DIV and TIMES constraints receives a "REAL" type value from the TIMES constraint, but already has "INTEGER" from the DIV constraint. A local context style error message

might be "Left expression of DIV must be an integer", but with backtracking turned on, the system could tell the user that "the left expression of the DIV operator must be of type integer, problem results from the use of the real constant '6.2'".

When the system detects an error, the location of the error is added to an error list. (In addition, each constraint and node should contain an error status flag, checkable by the pretty printer). When the error is fixed, its entry is removed from the error list. Although the user is not obliged to fix his errors immediately, should he attempt to leave the editor, the system can remind him that his program contains errors.

4.8 Summary

Although our design achieves the goals of handling the name binding, syntax and type checking of program fragments in an editing environment, it does not do all of its tasks as efficiently as it could. We do not seem to have entirely lived up to the philosophy of "doing as much as you can as soon as you can". As we have already noted, the problem stems from the inability of procedures to do local name

binding until they have received their container's environment. Rectifying this problem is the goal of chapter five.

5. THE IMPROVED DESIGN

In this chapter we detail the modifications required to make our original design more efficient and useful. We will achieve this by improving various aspects of name binding.

First, we wish to remove the environment dependencies between the declarations and between the statements in a procedure. Currently, all of the local declarations and statements of a procedure are chained together in a list. Thus, modification to any one of these declarations causes the environment to be retracted from all subsequent declarations in the list, from all of the statements of the procedure, and from all of the contained procedures. Any modification to a statement causes the environment in all subsequent statements in the procedure to be retracted. The amount of retraction and repropagation and the rebinding of every usage identifier in the procedure and in all contained procedures is obviously excessive. Most of the usage identifiers which have to be rebound will probably be bound to the same declaration!

Considering only the environment network, we propose to make each declaration in a procedure independent of its brother declarations. Any modification to a single

declaration will not cause retraction of any environment information other than from the modified declaration. Only those usage identifiers bound to this declaration will be affected. We propose to do the same for lists of statements as well. Thus, retraction and repropagation will be kept to a minimum.

Secondly, as mentioned before, we would like name binding to be able to take place inside a stand alone procedure; i.e. to make it unnecessary to wait until a path in the network is provided from the procedure to the root PROGRAM constraint. Currently, the internal environment is dependent upon an external environment. We propose to make the internal environment independent.

Finally, we wish to establish explicit bindings between usage identifiers and their declarations. Currently, the type and the assignability of an identifier can only be determined by the lookup procedure. Although a lookup will still have to be done when a usage identifier is first entered into the constraint network, instead of just looking up the declaration to find the type and the assignability values, we now propose to create a path from every usage identifier to its declaration consisting of type pins and assignment attribute pins. Hence, any subsequent modification to the type or the assignability of that identifier will cause proper updating in the network via the

usual retraction and propagation processes. No look up will be required and the environment network will remain untouched.

Since all of the required modifications are to do with name binding the only constraints in the network that will need to be redesigned are the constraints that appear in declarations (namely the PROGRAM VAR, PROCEDURE, and the type constraints); and, of course, the USAGE IDENTIFIER constraint. All of the other constraints described in chapter 4 will remain unchanged. These include the statement constraints, the operator constraints, and the constant constraints. The syntax, type, and assignability rules of these constraints also remain unchanged. So, for the purpose of our current discussion we can view a procedure or a program as consisting merely of a set of identifier declarations and a set of identifier usages.

With the rules of name binding in mind (from the scenarios described in Section 4.5) let us plunge into the improved design.

5.1 Removal of Environment Dependencies

In the simple design the PROGRAM and PROCEDURE

constraints had a "Name" branch and a "Body" branch. The "Body" branch was connected to a linear chain of declaration constraints which was followed by a linear chain of statement constraints.

The first step will be to remove the dependency between the declarations and statements. We do this by replacing the single "Body" branch of the PROGRAM and PROCEDURE constraints with two branches: the "Local Declarations" branch and the "Statements" branch. This modification to the local declarations will NOT cause any environment retraction to take place in the statements and vice versa.

The next step is to remove the environmental dependencies between the individual declarations in a declaration list, and the environmental dependencies between individual statements in a statement list. We must be careful how we proceed however. While we wish the individual elements in these lists to be independent in the environment network, we do NOT wish them to be independent in the syntax network. The order in which declarations and statements are entered by a user is important and cannot be tampered with. Our editor must never switch the order of consecutive statements or declarations. In the case of statements, the results could be disastrous.

The implication is clear. We must separate the syntax

network from the environment network. (The two networks currently run in parallel).

Our proposal is this: we will leave the syntax network as it stands and make the following changes to the environment network. When a VAR or PROCEDURE constraint is instantiated and inserted into a list of declarations, its "Parent" syntax pin will be attached to the preceding declaration and its "Next" syntax pin will be attached to the following declaration (as in the simple design), but its "Parent" environment pin will be attached directly to the "Local Declarations" environment pin of the local PROCEDURE constraint.

Similarly for statements: Lists of statements can occur wherever the "BEGIN ... END" construct of PASCAL can; i.e. as the body of a procedure or LOOP construct; or as the THEN or ELSE clause of an IF statement. We will call the constraint that contains the list of statements the "head" constraint. When inserting a new statement into a list of statements we attach the "Parent" and "Next Statement" syntax pins as before, but we attach the "Parent" environment pin of the statement constraint directly to the environment pin of the head constraint.

Hence, the internal representation of a list of declarations such as

```

PROCEDURE Catastrophe; •
  VAR a: ...;
  VAR b: ...;
  VAR c: ...;
BEGIN
  ...
END;

```

would be

Syntax: _____
 Environment: - - - - -

Figure 5.1 The internal representation of a list of declarations.

As you can see, the node attached to the "Declaration" environment pin of the PROCEDURE constraint contains many cells. This is the first time we have made use of this feature of Steele's system. Up until now, the maximum number of cells in any node in our network has been two.

Notice that because the environment propagates downwards, the removal of a single declaration or even a sequence of declarations will NOT cause the environment to be retracted from ANY other declaration in the list.

Notice also that the network constructor now has to have special knowledge of declarations lists (and statement lists) since inserting (or clipping) a sequence of declarations will require visiting each declaration and attaching (or severing) its "Parent" environment pin.

The internal representation of a list of statements is identical. For instance

```
WHILE ...  
  BEGIN  
    IF ...;  
    ... := ...;  
    WHILE ...  
  END
```

would be represented as

Syntax: —————
 Environment: - - - - -

Figure 5.2 The internal representation of a list of statements.

Again, notice that the removal of a single statement from the list of statements inside the outer WHILE loop will not cause retraction of the environment from any of the other statements in the list.

5.2 Procedures Do Their Own Name Binding

As mentioned before, we would like a procedure to be able to name bind itself. The reason for this is to immediately provide the user with as much information as possible on the

correctness of his procedures.

Before, a PROCEDURE constraint would not start propagating its own local environment (a pointer to itself) until it had received its container's environment on its "Parent" environment pin. Now, however, we want a PROCEDURE constraint to start propagating its own environment immediately upon instantiation. We do this by making the "Local Declarations" and "Statements" environment cells of the PROCEDURE constraint constant cells. The value in each constant cell will be set to be a pointer to the PROCEDURE constraint upon instantiation. Hence, the local environment will be propagated out to the local declarations and statements right away. Usage identifiers in the statements which are locally declared can be bound immediately. Other usage identifiers will momentarily remain unbound.

5.3 Explicit Bindings

To establish a direct connection in the network from a declaration to its usages we will need to add an additional branch to the VAR, PROCEDURE and USAGE IDENTIFIER constraints.

To the VAR and PROCEDURE constraints we add a "Usages"

branch consisting of a type pin and an assignment attribute pin. A rule is added to the VAR constraint stating that if a value arrives on the type pin of the "Type" branch, it is propagated out on the type pin of the "Usages" branch. In the case of the PROCEDURE constraint, the type pin on the "Usages" branch is attached to a constant cell containing a pointer to the PROCEDURE constraint (the type of a PROCEDURE is simply a pointer to itself).

The cells on the "Usages" assignment attribute pins of both the VAR and PROCEDURE constraints are constant cells. The VAR's cell contains the value "TRUE" (you can assign into a variable). The PROCEDURE's cell contains the value "FALSE" (you can not assign into a procedure).

To the USAGE IDENTIFIER constraint we add a "Declaration" branch consisting of a type pin and an assignment attribute pin. The type rule states that if a type value is received on the "Declaration" type pin, then this value is propagated out on the "Parent" type pin. Similarly, the assignability rule states that if an assignability value is received on the "Declaration" assignment attribute pin, then this value is propagated out on the "Parent" assignment attribute pin.

So, when an environment value first arrives at a USAGE IDENTIFIER constraint, the lookup procedure is invoked to find the declaration for this identifier. As before, if the

search fails, we have an undeclared identifier. If the search succeeds, the lookup process will now connect the "Declaration" branch of the USAGE IDENTIFIER constraint directly to the "Usages" branch of the declaration constraint. (The network builder ensures that the corresponding pins in the branches are connected appropriately).

The type and the assignment attribute values associated with that declaration will now be propagated to the USAGE IDENTIFIER constraint which will then propagate them up through the constraint network as before.

The following points are of note:

- o One declaration constraint can be connected (via a node) to many USAGE IDENTIFIER constraints. This connection can cross an arbitrary number of environment boundaries.
- o If the type in a declaration is ever removed, retraction of the type from all of its usages will take place and all the type values in the network dependent upon these usages will be retracted as well. And of course, when a new type is installed, propagation will take place supplying the new type value to all of its

usages. No environment retraction takes place and no lookups are performed.

- o Our network no longer has a simple tree structure. The connection from a USAGE IDENTIFIER constraint to a declaration constraint creates a circuit in the network. This added branch, however does not contain syntax pins. Those branches in the network that do contain syntax pins still form a tree which is, in fact, a spanning tree of the network.
- o The connections between USAGE IDENTIFIER constraints and declaration constraints are created by constraint rules. Before, propagation of information caused rules to be triggered, but these rules were never involved in the process of network construction.

That a triggered rule might modify the constraint network was not discussed by Steele but it is certainly within the capabilities of his system. Rules in his system are merely invocations of LISP functions and the connection procedure is itself just another LISP function.

However, if propagation of information causes a modification in the network, we must ensure

that retraction of that same information causes the network to return to its original state; i.e. we must add rules that are triggered when a cell of a constraint has its value retracted. In particular, retraction of the environment from a USAGE IDENTIFIER constraint must cause its "Declaration" branch to be disconnected and retraction of the environment from a VAR or PROCEDURE constraint must cause its "Usages" branch to be dissolved; i.e we must disconnect this declaration from ALL of its usages.

5.4 Problems

In our attempt to make the editor more efficient by reducing the amount of propagation and retraction of environment data and the number of usage identifier look ups, we have introduced two major problems.

The first has to do with the insertion of new declarations. When a new declaration is added or the name in an existing declaration is changed, no environment data is retracted from any USAGE IDENTIFIER constraint. So how do we ensure that all usages which should now be bound to

this new declaration do, in fact, get bound? It would appear that the new declaration must itself find all of its usages and do the binding; i.e., include a rule in the VAR and PROCEDURE constraints to perform a search of the usages, the rule being triggered when the name cell of the constraint receives a value. (Before, only USAGE IDENTIFIER constraints had rules that performed name binding.) To find its usages, the new declaration would have to search the local statements of the procedure containing the declaration and the statement parts of all the contained procedures. This looks like a lot of work; in fact, about the same amount of work as retracting the environment from the local statements and all the contained procedures, and then repropagating the new environment causing all usage identifiers to look themselves up again. Why is it so difficult for a declaration to look up its usages, when it is easy for a usage to look up its declaration? Because all declarations are kept in lists and we always know where the start of the list is. What appears to be needed, then, is a list of all those usages in the current environment (i.e., the one in which the new declaration resides) or in any of its contained environments that are either unbound or bound to a declaration in an environment external to the current environment. Furthermore, if the usages were somehow grouped by name in the list, the search for the usages to be bound to a new

declaration could be made that much easier.

The second problem has to do with the explicit bindings; i.e., the direct connections between a declaration and its list of usages. Deleting a declaration is easy to handle. It involves dissolving all of the connections to the usages and then performing lookups on just those usages. Remember, the removal of a declaration does not cause environment retraction in the procedure containing it or in any contained procedure, hence we have to force the lookups of all usages bound to this declaration to be done at the time of the deletion. Deleting an individual usage is also easy. It involves disconnecting it from the declaration. In both cases the bindings are broken by modifying just one node in the network.

The problem arises when the binding is between a declaration in some local environment and a usage identifier occurring in a contained environment; i.e., a contained procedure. Suppose a user decides he wants to clip out the contained procedure. Before, this was done internally by disconnecting the "Parent" branch of the contained PROCEDURE constraint from the "Next" branch of the preceding declaration constraint. Since all flow of data was restricted to the path created by this connection, cutting this connection resulted in a complete retraction of any implicit bindings between the two fragments. This still

must be done (although the "Parent" syntax and environment pins could be connected to different constraints and so must be disconnected individually), but there are also other connections between the two PROCEDURE constraints, namely, the branches binding the external declaration to the internal usage identifiers.

These bindings must be severed when the user clips the contained procedure.

To do this efficiently, for each PROCEDURE constraint we need access to all of those usage identifiers whose declarations exist in external environments. Unfortunately, in general it is not enough to know that a usage identifier has its declaration in some containing (external) environment in order to sever its bindings.

Consider the diagram in figure 5.3. It represents three nested procedures; "Procedure_1" is defined inside "Procedure_2" which is defined inside "Procedure_3". The dashed lines delimit environments. Names of identifiers on the left side of the dotted line represent declarations, names of identifiers on the right side represent usages.

Figure 5.3 Three nested procedures.

"Procedure_1" contains a declaration of the identifier "C" and in the statement part of "Procedure_1" we have usages of the identifiers "A", "B", and "C". "Procedure_2" contains a declaration of the identifier "B" and uses the variable "A" in its statement part. "Procedure_3" contains a declaration of the identifier "A".

Now, suppose the user wishes to "clip out" "Procedure_1". Since the identifiers "A" and "B" are declared externally to "Procedure_1", severing their bindings would be correct. *

Consider what happens if the user wishes to clip out "Procedure_2". Since "Procedure_1" is contained in "Procedure_2", the clipping of "Procedure_2" takes "Procedure_1" along with it. Now, as before, we have to sever the bindings of usages which are externally declared. However, and this is most important, we do not wish to sever the bindings of ALL the usages inside the fragment being clipped that happen to be declared externally. We would sever the binding of the usage of "A" in "Procedure_2" and also the binding of the usage of "A" in "Procedure_1". We would NOT sever the binding of the usage of "B" in "Procedure_1" because, even though the declaration of "B" is external to "Procedure_1", the declaration of "B" is contained in the fragment being clipped. So, in general it is not sufficient to know that a usage identifier is declared externally to the environment in which it is located. We must also know, for each environment E, which usage identifiers in E, or in any environment contained within E, are either unbound or have declarations outside of E. For example, we need to know that "A" and "B" are declared externally to Procedure_1 and that "A" is declared externally to Procedure_2.

In the event that a collection of nested procedures is clipped, the severing of a usage identifier's bindings will depend on whether the environment in which the identifier is

declared is external to the clipped fragment or not.

Since ANY procedure can be clipped at ANY time, it appears that we need a list, for every environment, of all those usages contained within that are bound to declarations in external environments. This is the same conclusion we reached after examining the first problem.

The details of the solution to these two problems are presented in the next section.

5.5 Solutions

We concluded in the last section that in EVERY environment, we need a list of all those usages contained within that are either unbound or externally bound. So, in addition to the list of local declarations that we currently maintain in every procedure, we propose to also maintain a list of external declarations. These are not real declarations as they do not have bodies of procedures or type definitions attached to them; we will refer to them hereafter as "pseudo declarations".

If a usage in some environment does not have a local declaration to bind itself to, then we bind it to a pseudo

declaration in the external declarations list. Suppose that in some environment we have a usage of an identifier, say "X". If there is no local declaration for "X", we conclude that the declaration for "X" appears in some external environment or does not exist at all. To reflect this, we place a pseudo declaration for "X" on the external declaration list in this environment. We can now bind the usage of "X" to this pseudo declaration. Furthermore suppose this environment is contained within another environment, and that this environment does not have a local declaration for "X" either. Again, we conclude that the real declaration for "X" must be in some containing environment or that it does not exist, so again, we place a pseudo declaration for "X" on the external declaration list in this environment and we bind it to the pseudo declaration for "X" that we just placed in the contained environment.

Continuing in this fashion, we create a chain of bindings from a usage identifier through a series of pseudo declarations and ultimately to a real local declaration for this identifier (if one exists). Hence, a local declaration can be bound directly to local usages and to one pseudo declaration in each of the immediately contained environments, but never directly to any usage in any contained environment; i.e. single connections in the network never cross more than one environment boundary at a

time.

So, if a usage is connected to a local declaration, either directly, or via a chain of pseudo declarations, the type and the assignability attribute of that identifier can propagate from that declaration to the usage.

Now, let us investigate more closely the nature of these pseudo declarations. In order to connect a list of external declarations to a PROCEDURE constraint, we must add an additional branch to it, called the "External Declarations" branch which consists of a single environment pin. When the PROCEDURE constraint receives an environment value on its "Parent" pin, it will propagate this value out on its "Name" environment pin and its "External Declarations" environment pin.

Attached to the "External Declarations" pin can be an arbitrary number of PSEUDO DECLARATION constraints. A PSEUDO DECLARATION constraint has four branches, the "Parent" branch (consisting of a single environment pin) is used to connect the PSEUDO DECLARATION to the PROCEDURE constraint, the "Name" branch is connected to a DEFINITION IDENTIFIER constraint (as in a local declaration), the "Declaration" and the "Usages" branches, both consisting of type and assignment attribute pins, are used to form the chain from a usage to its declaration.

Notice that there are no syntactic connections between any of the PSEUDO DECLARATION constraints. This reflects the fact that the user has no control over the creation, deletion and use of these pseudo declarations. It also reflects the fact that we are not concerned with the ordering of pseudo declarations within an environment.

The instantiation and addition of a PSEUDO DECLARATION constraint to a list of external declarations occurs during the LOOKUP operation.

As before, when an environment arrives on the parent pin of a USAGE IDENTIFIER constraint it is triggered into doing a lookup. We first search the list of local declarations for a declaration with the same name as the usage identifier. If we find one, then we bind the usage directly to the local declaration. However, if we do not find one, rather than looking in the container's environment for the declaration, instead, we search the list of external declarations in this local environment. If it is found here, we bind the usage's "Declaration" branch to the PSEUDO DECLARATION's "Usages" branch and we are done.

If we do not find it here, we must create a pseudo declaration for this identifier and add it to the list of external declarations. The USAGE IDENTIFIER is then bound to this new declaration. Having accomplished this, the

lookup procedure is done. However, we do not yet have a complete chain from the USAGE IDENTIFIER constraint to its declaration. The next step in building this chain is carried out by the new PSEUDO DECLARATION constraint.

A PSEUDO DECLARATION constraint is added to the list of external declarations by connecting its "Parent" environment pin to the "External Declarations" pin of the PROCEDURE constraint. If this PROCEDURE constraint is contained in some environment, then the PROCEDURE constraint will have propagated this environment as a value out on its "External Declarations" environment pin. This value is now propagated to the new PSEUDO DECLARATION causing its environment rule to be triggered. This rule is identical to the environment rule of a USAGE IDENTIFIER constraint. Thus a lookup is performed in the environment whose value was received on the "Parent" pin; in this case the container's environment.

This process repeats itself until a local or a pseudo declaration with the right name is found, or a newly added PSEUDO DECLARATION constraint does not receive an environment value on its parent pin; i.e., it is in an environment not contained within another environment. In the latter case, since the "Declaration" branch of the newly added PSEUDO DECLARATION constraint cannot be connected to a higher level declaration, the identifier associated with this PSEUDO DECLARATION is undeclared.

In the event that a chain of bindings IS created from a local declaration to a usage, the declaration can now propagate its type and its assignment attribute values out along the chain. The rules in the PSEUDO DECLARATION constraint state that if either a type value or an assignment attribute value is received on the "Declaration" branch, the same value is propagated out on the "Usages" branch.

Notice that the lookup procedure searches precisely one environment and that the connection between a "Declaration" branch and a "Usages" branch crosses at most one environment boundary. Furthermore, if a procedure contains many usages of an external variable (or procedure), the overall time for the lookup is greatly reduced. In the previous design, the time required to lookup each usage was the same, namely " $O(n)$ " where " n " is the number of environments searched.⁽¹⁾ In this design, the time required to build the chain from the first usage of the variable to its declaration is " $O(n)$ "; however, once the chain is built, the time required to lookup any other usage is a constant.

"Wow, that's great!", yelled Joe Tuba.

Figure 5.4 is an example of the new design showing the

(1) This is not strictly correct, since the time required to lookup each usage is also a function of the number of declarations in a local declarations list.

chain of bindings from usages to their declarations. In the figure, we have divided each environment into three regions separated by dotted lines. The left region contains local declarations, the upper right contains the external declarations, and the bottom right contains usages. Again, notice that a binding (represented as a double line) never crosses more than one environment boundary (the dashed lines) at a time.

Figure 5.4 Name bindings in the new design.

Figure 5.5 shows the new structure of a PROCEDURE constraint.

Syntax: —————
 Environment: - - - - -

Figure 5.5 The new PROCEDURE constraint.

So, our new design improves the lookup procedure, but what about those two problems we are supposed to be solving, namely: the severing of bindings of usages in a clipped procedure and binding a new declaration to all of its usages.

The first problem is easy to solve. Whenever a procedure is clipped out of a program, we must sever all of the pin connections between the clipped procedure and the remaining program fragment. A feature of our new design is that all of these connections can easily be found. They are the "Declaration" branches of the PSEUDO DECLARATION constraints

contained in the list of external declarations in the procedure being clipped. When the procedure is clipped out of the program, the environment value on the "Parent" pin of the PROCEDURE constraint is retracted. Since this value was originally propagated out to all of the PSEUDO DECLARATION constraints in the external declarations list, the value will now be retracted from each of these constraints. To achieve the desired results, when an environment is retracted from a PSEUDO DECLARATION constraint, it must disconnect its "Declaration" branch. If a PSEUDO DECLARATION had been connected to a real declaration, the type and assignment attribute values will be retracted from all of the usages attached to this PSEUDO DECLARATION, which are precisely those usages found in the clipped procedure and all of its contained procedures.

So, the desired affect is automatic. The usual propagation and retraction processes take care of everything. Extensive tree searches for usage identifiers are not required and only those usages that actually should be affected get affected, hence the minimum amount of work required is done.

Binding a new declaration to all of its usages is not quite as easy. When an attempt to add a new declaration to a list of local declarations is made, we must consider three possibilities.

One, there is already a local declaration with the same name. In this case, we must signal an error. ,

Two, there is neither a local declaration, nor a pseudo declaration with this name in this environment. In this case, we simply add the new declaration. No further work needs to be done, since there can be no usages of this identifier in this or any contained procedure (if so, there would have been a pseudo declaration for it, in this environment).

Three, there is no local declaration with this name, but there is a pseudo declaration with this name in this environment. If this is the case, then the PSEUDO DECLARATION constraint is already connected (via its "Usages" branch) to precisely those constraints to which the new local declaration should now be bound. So the surgical steps required are as follows. Disconnect the PSEUDO DECLARATION constraint's "Declaration" branch. This will cause the current type and assignment attribute values (if they exist) to be retracted from the constraint and hence from all usages dependent upon it. Next, attach the new local declaration constraint's "Usages" branch to the "Usages" branch of the PSEUDO DECLARATION constraint. This will cause the type and assignment attribute values of the new declaration to be propagated to all of its usages. Finally, disconnect the PSEUDO DECLARATION's "Parent"

environment pin and its "Usages" branch.(1)

The new design was developed to solve the problems of procedure clipping and declaration insertions. However, we now need a new method for handling the deletion of local declarations, which is essentially the inverse of the problem of inserting a new local declaration. This method is described below.

If the declaration we are about to delete is not connected to any usages, we simply remove the declaration and we are done.

If there are usages, we must instantiate a new PSEUDO DECLARATION constraint; attach a DEFINITION IDENTIFIER constraint to it with the same name as the identifier we are about to delete; connect the "Usages" branch of the PSEUDO DECLARATION constraint to the "Usages" branch of the local declaration constraint; delete the local DECLARATION; and, finally, connect the PSEUDO DECLARATION's "Parent" pin to the PROCEDURE constraint in this environment. The last action will result in the propagation of an environment to the new PSEUDO DECLARATION, which will in turn trigger the PSEUDO DECLARATION to look itself up and thus the proper

(1) The PSEUDO DECLARATION constraint and its attached DEFINITION IDENTIFIER constraint are now floating as a stand alone fragment invisible to the user. We should now return the fragment to an "available storage pool" for later re-use.

bindings to the external declaration (if it exists) will be established.

There remains one minor problem. If all usages are detached from a PSEUDO DECLARATION there is no longer any need for it to exist. Unfortunately, the usual propagation and retraction mechanisms do not provide us with a solution since the usages are dependent upon the PSEUDO DECLARATION and not vice-versa. The simplest solution is the laziest one; leave it in place. If the editor determines that it is running low on available memory, it can traverse the program fragments and remove any such PSEUDO DECLARATIONS.

"Well, that about raps it up!", exclaims Joe Catastrophe.

5.6 Summary

In this chapter, we have made several improvements to the design of the constraint network. We have removed the environment dependencies between individual declarations and statements. We have also removed the dependencies between procedures and their containers. Thus procedures are more modular, as they can now do much of their own name binding. Furthermore, because of the new way in which we bind usages to their declarations, lookups and the removal and insertion

of procedure and variable declarations have all been made more efficient.

6. SUMMARY AND CONCLUSIONS

At the outset, it was our desire to design an editor that was capable of enforcing programming rules of the following four classifications:

1. Syntax rules.
2. Type compatibility rules.
3. Assignability rules. (Those rules specifying which objects may have values assigned into them.)
4. Name binding rules. (Those rules specifying how variable and procedure usages and their declarations are to be bound.)

This thesis has shown how the constraint system developed by Steele and Sussman can be used to implement such an editor. We feel that the constraint formalism is a particularly appropriate design basis for a program editor for a variety of reasons.

- o A constraint computes using only locally available information; no global information is

required.

- o Since a constraint is not prejudiced as to the direction of the flow of data, it is capable of performing more deductions sooner than systems with predetermined data flow.
- o The method of dependency directed backtracking provides superior error tracing capabilities. The system is able to explain how and why it has arrived at its conclusions.
- o The system allows a user to incrementally construct his program and to be continuously aware of the correctness of his program. In fact, the system can detect errors in a stand alone fragment as small as "TRUE + 10".
- o Because almost all of the knowledge of the system is embedded in the rules of the constraints we achieve a high degree of modularity. The implementation of an editor for a similar programming language would require only minor modifications to the rules associated with some of the constraints. An editor for a language not including all the features of our toy language would not require as complex a design as the one presented in this paper. For

example, a language such as "C" does not allow nested procedures, hence the mechanism of pseudo declarations may not be needed.

- o This system has reasonable space requirements. The space required is on the order of the number of constraints in the network; i.e., a linear function of the program size.
- o The system is time efficient. As the internal network representing the user's program grows, information can propagate into those cells not already containing values. Cells already containing values do not have to be recomputed, hence each editing operation is accomplished with the minimum of effort. Only those cells which need to be affected are affected.

The principle challenge of the thesis was to design an editor that, within reasonable limits, restricted itself to the constraint features presented in Steele's thesis, and operated efficiently. Our first attempt, described in Chapter 4, adhered strictly to the basic constraint system. As we saw, this resulted in a prohibitively slow system. For every action, there was not necessarily an equal and opposite reaction. Even simple editing operations could result in massive amounts of retraction and repropagation.

We concluded that constraints could still be used by extending Steele's system to include retraction rules, and by improving our design (by removing unnecessary dependencies). Thus, we were able to produce a truly efficient editor which we presented in Chapter 5.

6.1 Extensions

While the toy language used in our system was adequate to exhibit many of the features commonly found in today's structured programming languages, it is missing many of the frills. We believe that it would be easy to include enumerated and subrange types; pointer, record and file types; user defined declarations of types, constants and functions; and the use of parameters in procedures and functions.

In addition, it would be interesting to investigate a generalization of the pseudo-declaration idea developed in chapter 5. We placed pseudo-declarations in every procedure to provide rapid access (hence faster propagation of information) to the undeclared usages contained within those environments. Perhaps the same device could be used in arbitrary program fragments to minimize the amount of work

required during clipping and pasting operations. Remember our motto: "Do as much as you can, as soon as you can".

Another area worthy of investigation is the automatic inferencing of types. In program fragments, the type of a variable can often be determined by its use. Consider "a := TRUE"; by the nature of the ASSIGNMENT constraint, the type of the variable "a" is CONSTRAINED to be boolean. Rather than waiting for the user to supply a declaration for each of his variables to the system, the system, in some cases, could actually install the required declaration for the user!

A constraint does not always have sufficient local knowledge to determine the exact type of a variable. In a case such as "a + 5", the PLUS constraint can only narrow the type down to be one of a set of possibilities; here, to real or integer. Assuming we could actually propagate sets of values, then by a process of elimination, the editor could eventually deduce the single correct type value. Again, this is just another example of trying to do as much for the user as soon as we can.

6.2 Applications and Future Reserach

We believe that the program editor we have developed in this thesis could be integrated into a complete programming system. The internal representation of abstract syntax trees as active constraint networks is particularly amenable to the addition of several programming system components. For example, the constraints of a program network could have additional rules which when triggered by propagation would cause the program to pretty print itself. Other system components which could be integrated in a similar fashion are program interpreting and incremental compiling. The latter has particular potential. When a user of our editor has succeeded in creating an error free program, the usual compiler tasks of parsing, type checking, and name binding are completed. Address computation and code generation are all that remain to be done in the translation from source language to object code. So machine language generation could simply be another component of our editor.

APPENDIX

The `DEFINE_CONSTRAINT` declaration is used to define a class of constraints. Apart from the extensions described in the thesis and below, this is essentially the same as Steele's `"DEFPRIM"` macro.

For instance, `"[DEFINE_CONSTRAINT If ...]"` defines the class of `"If"` constraints. An instance of such a constraint may be created with a `"(CREATE If-46 If)"` request. In this particular case an `If` constraint called `"If-46"` will be created.

`DEFINE_CONSTRAINT` has the following structure:

```

[DEFINE CONSTRAINT <name>
  [PINS
    (<pin_description>)
    .
    .
    . ]

  [CAN_BE_ONE_OF
    (<pin_name> (list of values))
    .
    .
    . ]

  [MUST_CONNECT list of branch names]

  [EXTRAS
    <variable> ...]

  [INITIALIZE
    <initialization>
    .
    .
    . ]

  [<Rules_for_some_network> {eg. Environment_Rules}
    [<Rules>
    .
    .
    . ]]

]

```

The "PINS" clause lists the pins attached to the constraint. The pins may be simple pins as in Steele's constraint system or "structured pins". A structured pin is a branch name with a list of component pin names, eg. "Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin)". This means that the Expression branch consists of four pins which we can refer to as

Expression.Syntax_Pin, Expression.Environment_Pin,
Expression.Type_Pin and Expression.Assign_Attribute_Pin.

We find this description useful, because on the one hand, when the Editor is required to wire together two Expressions (for example) it really means that it has to wire together each of the four "sub-pins", and on the other hand it is important to be able to talk about what is being propagated through the different networks; i.e., the Type network, the Environment network, etc. Also it is convenient to define the rules for each constraint in terms of which network they affect; i.e., Type_Rules, Syntax_Rules etc.

The CAN_BE_ONE_OF clause is treated as a comment, we include it here for completeness and hopefully, as an aid to understanding what is being propagated through the program tree networks. It merely lists all of the pins of the given constraint with a list of their possible values (or description of their possible values; "?" implies that the value can only be determined at run time.

The "MUST_CONNECT" clause specifies those branches of a constraint which must be connected before the network is considered to be complete; i.e., the network does not constitute a complete program until every "MUST_CONNECT" branch has actually been connected to some other branch in the network.

The "EXTRAS" clause allows us to define additional variables to be associated with a given constraint. These are not pins. They cannot be wired into any other constraint. They are merely repositories for storing useful information. For instance, in the case of the "Integer_Constant" constraint we need a variable to store the actual value of the integer constant.

The "INITIALIZE" clause allows us to set the values in the cells (which are at the ends of the pins, remember) or the values in the EXTRAS. These assignments will be done at the time of the creation of an instance of the given constraint.

Finally, the rule clauses define what actions are to take place when new values appear on the pins of a constraint. Each rule has a list of "TRIGGERED_BY" pins which must have values for the rule to be invoked.

Simple Design Constraints

The following are the definitions of the constraints used in the simple design.


```

[DEFINE_CONSTRAINT Program_Constraint

[PINS
  (Parent (Environment_Pin))
  (Name (Syntax_Pin Environment_Pin Actual_Name_Pin))
  (Body (Syntax_Pin Environment_Pin))]

[CAN_BE_ONE_OF
  (Parent.Environment_Pin "STANDARD_ENVIRONMENT")
  (Name.Syntax_Pin Definition_Identifier)
  (Name.Environment_Pin "STANDARD_ENVIRONMENT")
  (Name.Actual_Name ?)
  (Body.Syntax_Pin (Definition_Statement))
  (Body.Environment_Pin "ME")]

[MUST_CONNECT Name Body]

[EXTRAS Assignment_Attribute]

[INITIALIZE
  (SET_EXTRA Assignment_Attribute FALSE)
  (CONSTANT_CELL Parent.Environment_Pin "STANDARD_ENVIRONMENT")
  (CONSTANT_CELL Name.Environment_Pin "STANDARD_ENVIRONMENT")
  (CONSTANT_CELL Name.Syntax_Pin Definition_Identifier)]

[Syntax_Pin_Rules
  [(TRIGGERED_BY Body.Syntax_Pin)
   (SELECT
    [(Body.Syntax_Pin = Declaration) OK]
    [(Body.Syntax_Pin = Statement) OK]
    [OTHERWISE (ERROR 'Need declaration or statement')])]
   Environment_Rules
  [(TRIGGERED_BY Name.Actual_Name_Pin)
   (SET_CELL Body.Environment_Pin "ME")]]]

[OFFLINE_CONSTRAINT Procedure_Constraint

[PINS
  (Parent (Syntax_Pin Environment_Pin))
  (Name (Syntax_Pin Environment_Pin Actual_Name_Pin))

```

```

      (Body (Syntax_Pin Environment_Pin))
      (Next (Syntax_Pin Environment_Pin)))

[CAN_BE_ONE_OF
  (Parent_Syntax_Pin Declaration)
  (Parent_Syntax_Pin Container's_environment)
  (Name_Syntax_Pin Declaration_Identifier)
  (Name_Environment_Pin Container's_environment)
  (Name_Actual_Name_Pin ?)
  (Body_Syntax_Pin (Declaration Statement))
  (Body_Environment_Pin "NE")
  (Next_Syntax_Pin (Declaration Statement))
  (Next_Environment_Pin "NE")]

[MUST_CONNECT Parent Name Body]

[EXTRAS Assignment_Attribute]

[INITIALIZE
  (SET_EXTRAS Assignment_Attribute FALSE)
  (CONSTANT_CELL Parent_Syntax_Pin Declaration)
  (CONSTANT_CELL Name_Syntax_Pin Declaration_Identifier)]

[Syntax_Pin_Rules
  [(TRIGGERED_BY Body_Syntax_Pin)
   (SELECT
     [(Body_Syntax_Pin = Declaration) OK]
     [(Body_Syntax_Pin = Statement) OK]
     [OTHERWISE (ERROR 'Need declaration or statement')])]
  [(TRIGGERED_BY Next_Syntax_Pin)
   (SELECT
     [(Next_Syntax_Pin = Declaration) OK]
     [(Next_Syntax_Pin = Statement) OK]
     [OTHERWISE (ERROR 'Need declaration or statement')])]
  [Environment_Rules
   [(TRIGGERED_BY Parent_Environment_Pin)
    (SET_CELL Name_Environment_Pin Parent_Environment_Pin)]
   [(TRIGGERED_BY Name_Actual_Name_Pin)
    (SET_CELL Next_Environment_Pin Parent_Environment_Pin)]]

```

```

      (SET_CELL Body.Environment.Pin "ME"))]]

[DEFINE_CONSTRAINT Var_Constraint

[PINS
  (Parent (Syntax_Pin Environment_Pin))
  (Name (Syntax_Pin Environment_Pin Actual_Name_Pin))
  (Type (Syntax_Pin Environment_Pin Type_Pin))
  (Next (Syntax_Pin Environment_Pin))]]

[[CAN_BE_ONE_OF
  (Parent, Syntax_Pin Declaration)
  (Parent, Environment_Pin ?)

  (Name, Syntax_Pin Declaration_Identifier)
  (Name, Environment_Pin ?)
  (Name, Actual_Name_Pin ?)

  (Type, Syntax_Pin Type)
  (Type, Environment_Pin ?)
  (Type, Type_Pin ?)

  (Next, Syntax_Pin (Declaration Statement))
  (Next, Environment_Pin ?)]

[MUST_CONNECT Parent Name Type]

[EXTRAS Assignment_Attribute]

[INITIALIZE
  (SET EXTRA Assignment_Attribute TRUE),

  (CONSTANT_CELL Parent, Syntax_Pin Declaration)
  (CONSTANT_CELL Name, Syntax_Pin Declaration_Identifier)
  (CONSTANT_CELL Type, Syntax_Pin Type)]

[Syntax_Pin_Rules
  (TRIGGERED_BY Next, Syntax_Pin)
  (SELECT
    [(Next, Syntax_Pin = Declaration) OK]
    [(Next, Syntax_Pin = Statement) OK]
    [OTHERWISE (ERROR "Need declaration or statement'')]])]

[Environment_Rules
  [(TRIGGERED_BY Parent, Environment_Pin)

```

POOR PRINT
Epreuve illisible

```

      (SET_CELL Name.Environment.Pin Parent.Environment.Pin))
    [(TRIGGERED_BY Name.Actual_Name.Pin)
     (SET_CELL Type.Environment.Pin Parent.Environment.Pin))
    [(TRIGGERED_BY Type.Type.Pin)
     (SET_CELL Next.Environment.Pin Parent.Environment.Pin)]]]

[DEFINE_CONSTRAINT Declaration_Identifier_Constraint
 [PINS
  (Parent (Syntax_Pin Environment_Pin Identifier_Name))]
 [CAN_BE_ONE_OF
  (Parent.Syntax_Pin Declaration_Identifier)
  (Parent.Environment_Pin ?)
  (Parent.Identifier_Name ?)]
 [MUST_CONNECT Parent]
 ] [EXTRAS Actual_Name]
 : [INITIALIZE
  (CONSTANT_CELL Parent.Syntax_Pin Declaration_Identifier))
  (Environment_Rules
   [(TRIGGERED_BY Parent.Environment_Pin)
    (LET (Name_Already_Exists_In_This_Scope (LOOKUP_DEFINITION_IDENTIFIER Actual_Name Parent.Environment_Pin))
     (IF Name_Already_Exists_In_This_Scope
      THEN (ERROR Actual_Name " is already defined in the local environment")
      ELSE (SET_CELL Parent.Identifier_Name Actual_Name)))]])

[DEFINE_CONSTRAINT Integer_Type_Constraint
 [PINS
  (Parent (Syntax_Pin Environment_Pin Type_Pin))]
 [CAN_BE_ONE_OF
  (Parent.Syntax_Pin Type)
  (Parent.Environment_Pin ?)
  (Parent.Type_Pin "NE")]
 [MUST_CONNECT Parent]

```

```

[INITIALIZE
  (CONSTANT_CELL Parent.Syntax_Pin Type)]
[Environment_Rules
  [(TRIGGERED_BY Parent.Environment_Pin)
   (SET_CELL Parent.Type_Pin "ME")]]]

[DEFINE_CONSTRAINT Real_Type_Constraint

[PINS
  (Parent (Syntax_Pin Environment_Pin Type))]

[CAN_BE_ONE_OF
  (Parent.Syntax_Pin Type)
  (Parent.Environment_Pin ?)
  (Parent.Type_Pin "ME")]

[MUST_CONNECT Parent]

[INITIALIZE
  (CONSTANT_CELL Parent.Syntax_Pin Type)]
[Environment_Rules
  [(TRIGGERED_BY Parent.Environment_Pin)
   (SET_CELL Parent.Type_Pin "ME")]]]

[DEFINE_CONSTRAINT Boolean_Type_Constraint

[PINS
  (Parent (Syntax_Pin Environment_Pin Type))]

[CAN_BE_ONE_OF
  (Parent.Syntax_Pin Type)
  (Parent.Environment_Pin ?)
  (Parent.Type_Pin "ME")]

[MUST_CONNECT Parent]

[INITIALIZE
  (CONSTANT_CELL Parent.Syntax_Pin Type)]
[Environment_Rules
  [(TRIGGERED_BY Parent.Environment_Pin)
   (SET_CELL Parent.Type_Pin "ME")]]]

```

POOR PRINT
Epreuve illisible

```

[DEFINE_CONSTRAINT Array-Constructor_Type

[PINS
  (Parent (Syntax_Pin Environment_Pin Type_Pin))
  (Lower_Bound (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
  (Upper_Bound (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
  (Element_Type (Syntax_Pin Environment_Pin Type_Pin))]

[CAN_BE_ONE_OF
  (Parent_Syntax_Pin Type)
  (Parent_Environment_Pin ?)
  (Parent_Type_Pin "ME")

  (Lower_Bound_Syntax_Pin Constant)
  (Lower_Bound_Environment_Pin ?)
  (Lower_Bound_Type_Pin "Integer_Type_Constraint")
  (Lower_Bound_Assign_Attribute_Pin ?)

  (Upper_Bound_Syntax_Pin Constant)
  (Upper_Bound_Environment_Pin ?)
  (Upper_Bound_Type_Pin "Integer_Type_Constraint")
  (Upper_Bound_Assign_Attribute_Pin ?)

  (Element_Type_Syntax_Pin Type)
  (Element_Type_Environment_Pin ?)
  (Element_Type_Type_Pin ?)]

[MUST_CONNECT Parent Lower_Bound Upper_Bound Element_Type]

INITIALIZE
  (CONSTANT_CELL Parent_Syntax_Pin Type)
  (CONSTANT_CELL Lower_Bound_Syntax_Pin Constant)
  (CONSTANT_CELL Upper_Bound_Syntax_Pin Constant)
  (CONSTANT_CELL Element_Type_Syntax_Pin Type)

  (CONSTANT_CELL Lower_Bound_Type_Pin "Integer_Type_Constraint")
  (CONSTANT_CELL Upper_Bound_Type_Pin "Integer_Type_Constraint")

[Environment_Rules
  [(TRIGGERED_BY Parent_Environment_Pin)
   (SET_CELL Element_Type_Environment_Pin Parent_Environment_Pin)]]

[Type_Rules
  [(TRIGGERED_BY Element_Type_Pin)
   (SET_CELL Parent_Type_Pin "ME")]]

```

```

[DEFINE_CONSTRAINT If_Constraint
  IF THS
    (Parent (Syntax_Pin Environment_Pin))
    (Condition (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
    (Then (Syntax_Pin Environment_Pin))
    (Else (Syntax_Pin Environment_Pin))
    (Next_Statement (Syntax_Pin Environment_Pin))]

[CAN_BE_ONE_OF
  (Parent_Syntax_Pin Statement)
  (Parent_Environment_Pin ?)
  (Condition_Syntax_Pin Expression)
  (Condition_Environment_Pin ?)
  (Condition_Type_Pin Boolean_Type_Constraint)
  (Condition_Assign_Attribute_Pin ?)
  (Then_Syntax_Pin Statement)
  (Then_Environment_Pin ?)
  (Else_Syntax_Pin Statement)
  (Else_Environment_Pin ?)
  (Next_Statement_Syntax_Pin Statement)
  (Next_Statement_Environment_Pin ?)]

[MUST_CONNECT Parent Condition Then]

[INITIALIZE
  (CONSTANT_CELL Parent_Syntax_Pin Statement)
  (CONSTANT_CELL Condition_Syntax_Pin Expression)
  (CONSTANT_CELL Then_Syntax_Pin Statement)
  (CONSTANT_CELL Else_Syntax_Pin Statement)
  (CONSTANT_CELL Next_Statement_Syntax_Pin Statement)
  (CONSTANT_CELL Condition_Type_Pin Boolean_Type_Constraint)]

[Environment_Rules
  [(TRIGGERED_BY Parent_Environment_Pin Parent_Environment_Pin)
   (SET_CELL Condition_Environment_Pin Parent_Environment_Pin)
   (SET_CELL Then_Environment_Pin Parent_Environment_Pin)
   (SET_CELL Else_Environment_Pin Parent_Environment_Pin)
   (SET_CELL Next_Statement_Environment_Pin Parent_Environment_Pin)]]]

```

```

[DEFINE_CONSTRAINT While_Constraint
[PINS
  (Parent (Syntax_Pin Environment_Pin))
  (Condition (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
  (Body (Syntax_Pin Environment_Pin))
  (Next_Statement (Syntax_Pin Environment_Pin))]
[CAN_BE_ONE_OF
  (Parent, Syntax_Pin Statement)
  (Parent, Environment_Pin ?)
  (Condition, Syntax_Pin Expression)
  (Condition, Environment_Pin ?)
  (Condition, Type_Pin "Boolean_Type_Constraint")
  (Condition, Assign_Attribute_Pin "don't_care")
  (Body, Syntax_Pin Statement)
  (Body, Environment_Pin ?)
  (Next_Statement, Syntax_Pin Statement)
  (Next_Statement, Environment_Pin ?)]
[MUST_CONNECT Parent Condition_Body]
[INITIALIZE
  (CONSTANT_CELL Parent, Syntax_Pin Statement)
  (CONSTANT_CELL Condition, Syntax_Pin Expression)
  (CONSTANT_CELL Body, Syntax_Pin Statement)
  (CONSTANT_CELL Next_Statement, Syntax_Pin Statement)
  (CONSTANT_CELL Condition, Type_Pin "Boolean_Type_Constraint")]
[Environment_Rules
  [(TRIGGERED_BY Parent, Environment_Pin)
   (SET_CELL Condition, Environment_Pin Parent.Environment_Pin)
   (SET_CELL Body, Environment_Pin Parent.Environment_Pin)
   (SET_CELL Next_Statement, Environment_Pin Parent.Environment_Pin)]]]
[DEFINE_CONSTRAINT Assignment_Constraint
[PINS
  (Parent (Syntax_Pin Environment_Pin))
  (Variable_Reference (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
  (Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))

```



```

        (Next_Statement      (Syntax_Pin Environment_Pin))]]

[CAN_BE_ONE_OF
  (Parent_Syntax_Pin      Statement)
  (Parent_Environment_Pin ?)

  (Variable_Reference_Syntax_Pin      Expression)
  (Variable_Reference_Environment_Pin ?)
  (Variable_Reference_Type_Pin      ("Integer_Type_Constraint "Real_Type_Constraint "Boolean_Type_Constraint "Array_Constructor_Type_Constraint"))
  (Variable_Reference_Assign_Attribute_Pin TRUE)

  (Expression_Syntax_Pin      Expression)
  (Expression_Environment_Pin ?)
  (Expression_Type_Pin      ("Integer_Type_Constraint "Real_Type_Constraint "Boolean_Type_Constraint "Array_Constructor_Type_Constraint"))
  (Expression_Assign_Attribute_Pin don't_care)

  (Next_Statement_Syntax_Pin      Statement)
  (Next_Statement_Environment_Pin ?)]

[MUST_CONNECT Parent Variable_Reference Expression]

[INITIALIZE
  (CONSTANT_CELL Parent_Syntax_Pin      Statement)
  (CONSTANT_CELL Variable_Reference_Syntax_Pin Expression)
  (CONSTANT_CELL Expression_Syntax_Pin      Expression)
  (CONSTANT_CELL Next_Statement_Syntax_Pin      Statement)
  (CONSTANT_CELL Variable_Reference_Assign_Attribute_Pin TRUE)]

[Environment_Rules
  [ (TRIGGERED_BY Parent_Environment_Pin)
    (SET_CELL Variable_Reference_Environment_Pin Parent_Environment_Pin)
    (SET_CELL Expression_Environment_Pin      Parent_Environment_Pin)
    (SET_CELL Next_Statement_Environment_Pin      Parent_Environment_Pin)]]

[Type_Rules
  [ (TRIGGERED_BY Expression_Type_Pin)
    - (IF (Expression_Type_Pin = "Integer_Type_Constraint")
      THEN OK
      ELSE (SET_CELL Variable_Reference_Type_Pin Expression_Type_Pin))]

  [ (TRIGGERED_BY Variable_Reference_Type_Pin)
    (IF (Variable_Reference_Type_Pin = "Real_Type_Constraint")
      THEN OK
      ELSE (SET_CELL Expression_Type_Pin Variable_Reference))]]

```

```

[ (TRIGGERED_BY Variable_Reference.Type_Pin Expression.Type_Pin)
  (IF (Variable_Reference.Type_Pin = 'Real_Type_Constraint')
    THEN
      (IF (OR (Expression.Type_Pin = 'Integer_Type_Constraint') (Expression.Type_Pin = 'Real_Type_Constraint'))
        THEN OK
        ELSE (ERROR 'Can only assign real or integer expressions into a variable of type real'))
      ELSE OK)]]]

```

[DEFINE_CONSTRAINT Call_Constraint

```

  [PINS
    (Parent (Syntax_Pin Environment_Pin))
    (Procedure_Name (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
    (Next_Statement (Syntax_Pin Environment_Pin))]

```

[CAN_BE_ONE_OF

```

  (Parent.Syntax_Pin Statement)
  (Parent.Environment_Pin ?)
  (Procedure_Name.Syntax_Pin Useas_Identifier)
  (Procedure_Name.Environment_Pin ?)
  (Procedure_Name.Type_Pin 'Procedure_Constraint')
  (Procedure_Name.Assign_Attribute_Pin don't_care)
  (Next_Statement.Syntax_Pin Statement)
  (Next_Statement.Environment_Pin ?)]

```

[MUST_CONNECT Parent_Procedure_Name]

```

[INITIALIZE
  (CONSTANT_CELL Parent.Syntax_Pin Statement)
  (CONSTANT_CELL Procedure_Name.Syntax_Pin Useas_Identifier)
  (CONSTANT_CELL Next_Statement.Syntax_Pin Statement)
  (CONSTANT_CELL Procedure_Name.Type_Pin Procedure)]

```

[Environment_Rules

```

  [(TRIGGERED_BY Parent.Environment_Pin)
    (SET_CELL Procedure_Name.Environment_Pin Parent.Environment_Pin)
    (SET_CELL Next_Statement.Environment_Pin Parent.Environment_Pin)]]]

```

[DEFINE_CONSTRAINT Times_Constraint (the definitions for Binary_Plus and Binary_Minus are identical.

[PINS

```

(Result
  (Left_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
  (Right_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin)))
[CAN_BE_ONE_OF
  (Result_Syntax_Pin Expression)
  (Result_Environment_Pin ?)
  (Result_Type_Pin (~Integer_Type_Constraint ~Real_Type_Constraint))
  (Result_Assign_Attribute_Pin FALSE)
  (Left_Expression_Syntax_Pin Expression)
  (Left_Expression_Environment_Pin ?)
  (Left_Expression_Type_Pin (~Integer_Type_Constraint ~Real_Type_Constraint))
  (Left_Expression_Assign_Attribute_Pin don't_care)
  (Right_Expression_Syntax_Pin Expression)
  (Right_Expression_Environment_Pin ?)
  (Right_Expression_Type_Pin (~Integer_Type_Constraint ~Real_Type_Constraint))
  (Right_Expression_Assign_Attribute_Pin don't_care)]
[MUST_CONNECT Result Left_Expression Right_Expression]
[INITIALIZE
  (CONSTANT_CELL Result_Syntax_Pin Expression)
  (CONSTANT_CELL Left_Expression_Syntax_Pin Expression)
  (CONSTANT_CELL Right_Expression_Syntax_Pin Expression)
  (CONSTANT_CELL Result_Assign_Attribute_Pin FALSE)
  (Environment_Rules
    (SET_CELL Left_Expression_Environment_Pin Result_Environment_Pin)
    (SET_CELL Right_Expression_Environment_Pin Result_Environment_Pin))]
[Type_Rules
  ; If both the operands are of type integer then the result shall
  ; be of type integer, but if otherwise, the result shall be of
  ; type real.
  [(TRIGGERED_BY Result_Type_Pin)
   (SELECT
    [(Result_Type_Pin = Integer)
     (SET_CELL Left_Expression_Type_Pin ~Integer_Type_Constraint)
     (SET_CELL Right_Expression_Type_Pin ~Integer_Type_Constraint)]
    [(Result_Type_Pin = ~Real_Type_Constraint) OK]
    [OTHERWISE (ERROR "Result of times operation must be integer or real")]])]

```

```

[(TRIGGERED_BY Left_Expression.Type_Pin)
 (SELECT
  [(Left_Expression.Type_Pin = "Real_Type_Constraint") (SET_CELL Result.Type_Pin "Real_Type_Constraint")]
  [(Left_Expression.Type_Pin = "Integer_Type_Constraint") OK]
  [OTHERWISE (ERROR "Left operand of times operation must be integer or real")]))]

[(TRIGGERED_BY Right_Expression.Type_Pin)
 (SELECT
  [(Right_Expression.Type_Pin = "Real_Type_Constraint") (SET_CELL Result.Type_Pin "Real_Type_Constraint")]
  [(Right_Expression.Type_Pin = "Integer_Type_Constraint") OK]
  [OTHERWISE (ERROR "Right operand of times operation must be integer or real")]))]

[(TRIGGERED_BY Result.Type_Pin Left_Expression.Type_Pin)
 (SELECT
  [(AND (Result.Type_Pin = "Real_Type_Constraint") (Left_Expression.Type_Pin = "Real_Type_Constraint")) OK]
  [(AND (Result.Type_Pin = "Real_Type_Constraint") (Left_Expression.Type_Pin = "Integer_Type_Constraint"))
   (SET_CELL Right_Expression.Type_Pin "Real_Type_Constraint")]
  [(AND (Result.Type_Pin = "Integer_Type_Constraint") (Left_Expression.Type_Pin = "Integer_Type_Constraint"))
   (SET_CELL Right_Expression.Type_Pin "Integer_Type_Constraint")]
  [OTHERWISE (ERROR "Contradiction between types of result and left operand of times operation")]])]

[(TRIGGERED_BY Result.Type_Pin Right_Expression.Type_Pin)
 (SELECT
  [(AND (Result.Type_Pin = "Real_Type_Constraint") (Right_Expression.Type_Pin = "Real_Type_Constraint")) OK]
  [(AND (Result.Type_Pin = "Real_Type_Constraint") (Right_Expression.Type_Pin = "Integer_Type_Constraint"))
   (SET_CELL Left_Expression.Type_Pin "Real_Type_Constraint")]
  [(AND (Result.Type_Pin = "Integer_Type_Constraint") (Right_Expression.Type_Pin = "Integer_Type_Constraint"))
   (SET_CELL Left_Expression.Type_Pin "Integer_Type_Constraint")]
  [OTHERWISE (ERROR "Contradiction between types of result and right operand of times operation")]])]

[(TRIGGERED_BY Left_Expression.Type_Pin Right_Expression.Type_Pin)
 (SELECT
  [(AND (Left_Expression.Type_Pin = "Real_Type_Constraint") (Right_Expression.Type_Pin = "Real_Type_Constraint"))
   (SET_CELL Result.Type_Pin "Real_Type_Constraint")]
  [(AND (Left_Expression.Type_Pin = "Real_Type_Constraint") (Right_Expression.Type_Pin = "Integer_Type_Constraint"))
   (SET_CELL Result.Type_Pin "Real_Type_Constraint")]
  [(AND (Left_Expression.Type_Pin = "Integer_Type_Constraint") (Right_Expression.Type_Pin = "Real_Type_Constraint"))
   (SET_CELL Result.Type_Pin "Integer_Type_Constraint")]
  [(AND (Left_Expression.Type_Pin = "Integer_Type_Constraint") (Right_Expression.Type_Pin = "Integer_Type_Constraint"))
   (SET_CELL Result.Type_Pin "Integer_Type_Constraint")]
  [OTHERWISE (ERROR "Contradiction between types of times operation")]])]

```

[DEFINE_CONSTRAINT Real_Divide_Constraint

[PINS

3 3

OF / DE

```

(Result      (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
(Left_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
(Right_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))]]

[CAN_BE_ONE_OF
(Result, Syntax_Pin      Expression)
(Result, Environment_Pin ?)
(Result, Type_Pin      "Real_Type_Constraint")
(Result, Assign_Attribute_Pin FALSE)

(Left_Expression, Syntax_Pin      Expression)
(Left_Expression, Environment_Pin ?)
(Left_Expression, Type_Pin      ("Integer_Type_Constraint "Real_Type_Constraint"))
(Left_Expression, Assign_Attribute_Pin don't_care)

(Right_Expression, Syntax_Pin      Expression)
(Right_Expression, Environment_Pin ?)
(Right_Expression, Type_Pin      ("Integer_Type_Constraint "Real_Type_Constraint"))
(Right_Expression, Assign_Attribute_Pin don't_care))]

[MUST_CONNECT Result Left_Expression Right_Expression]

[INITIALIZE
(CONSTANT_CELL Result, Syntax_Pin      Expression)
(CONSTANT_CELL Left_Expression, Syntax_Pin      Expression)
(CONSTANT_CELL Right_Expression, Syntax_Pin      Expression)

(CONSTANT_CELL Result, Type_Pin "Real_Type_Constraint")

(CONSTANT_CELL Result, Assign_Attribute_Pin FALSE))]

[Environment_Rules
("TRIGGERED_BY Result, Environment_Pin)
 (SET_CELL Left_Expression, Environment_Pin Result, Environment_Pin)
 (SET_CELL Right_Expression, Environment_Pin Result, Environment_Pin))]

[Type_Rules
("TRIGGERED_BY Left_Expression, Type_Pin)
 (IF (OR (Left_Expression, Type_Pin = "Real_Type_Constraint) (Left_Expression, Type_Pin = "Integer_Type_Constraint))
  THEN OK
  ELSE (ERROR "Left operand of division must be integer or real"))]

("TRIGGERED_BY Right_Expression, Type_Pin)
 (IF (OR (Right_Expression, Type_Pin = "Real_Type_Constraint) (Right_Expression, Type_Pin = "Integer_Type_Constraint))
  THEN OK
  ELSE (ERROR "Right operand of division must be integer or real")))]

```

DEFINE_CONSTRAINT Integer_Divide_Constraint :the 'DIV' of PASCAL :the definition for MOD is identical.

```

CPINS
(Result
  (Left_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
  (Right_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin)))
(CAN_BE_ONE_OF
  (Result, Syntax_Pin Expression)
  (Result, Environment_Pin ?)
  (Result, Type_Pin "Integer_Type_Constraint")
  (Result, Assign_Attribute_Pin FALSE)
  (Left_Expression, Syntax_Pin Expression)
  (Left_Expression, Environment_Pin ?)
  (Left_Expression, Type_Pin "Integer_Type_Constraint")
  (Left_Expression, Assign_Attribute_Pin don't_care)
  (Right_Expression, Syntax_Pin Expression)
  (Right_Expression, Environment_Pin ?)
  (Right_Expression, Type_Pin "Integer_Type_Constraint")
  (Right_Expression, Assign_Attribute_Pin don't_care))
[MUST_CONNECT Result Left_Expression Right_Expression]

[INITIALIZE
  (CONSTANT_CELL Result, Syntax_Pin Expression)
  (CONSTANT_CELL Left_Expression, Syntax_Pin Expression)
  (CONSTANT_CELL Right_Expression, Syntax_Pin Expression)
  (CONSTANT_CELL Result, Type_Pin "Integer_Type_Constraint")
  (CONSTANT_CELL Left_Expression, Type_Pin "Integer_Type_Constraint")
  (CONSTANT_CELL Right_Expression, Type_Pin "Integer_Type_Constraint")
  (CONSTANT_CELL Result, Assign_Attribute_Pin FALSE)]

[Environment_Rules
  [TRIGGERED_BY Result, Environment_Pin]
  (SET_CELL Left_Expression, Environment_Pin Result, Environment_Pin)
  (SET_CELL Right_Expression, Environment_Pin Result, Environment_Pin)]]]

```

DEFINE_CONSTRAINT Unary_Plus_Constraint :the definition for Unary_Minus is identical.

```

CPINS
(Result
  (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))

```

```

(Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin)))
[CAN_BE_ONE_OF
  (Result_Syntax_Pin Expression)
  (Result_Environment_Pin ?)
  (Result_Type_Pin ("Integer_Type_Constraint "Real_Type_Constraint))
  (Result_Assign_Attribute_Pin FALSE)
  (Expression_Syntax_Pin Expression)
  (Expression_Environment_Pin ?)
  (Expression_Type_Pin ("Integer_Type_Constraint "Real_Type_Constraint))
  (Expression_Assign_Attribute_Pin don't_care)]
[MUST_CONNECT Result Expression]

INITIALIZE
  (CONSTANT_CELL Result_Syntax_Pin Expression)
  (CONSTANT_CELL Expression_Syntax_Pin Expression)
  (CONSTANT_CELL Result_Assign_Attribute_Pin FALSE))

[Environment_Rules
  [(TRIGGERED_BY Result_Environment_Pin)
   (SET_CELL Expression_Environment_Pin Result_Environment_Pin)]]

[Type_Rules
  [(TRIGGERED_BY Result_Type_Pin)
   (SELECT
    [(Result_Type_Pin = "Real_Type_Constraint) (SET_CELL Expression_Type_Pin "Real_Type_Constraint))
    [(Result_Type_Pin = "Integer_Type_Constraint) (SET_CELL Expression_Type_Pin "Integer_Type_Constraint))
    [OTHERWISE (ERROR "Result of unary plus operation must be integer or real')]]])
  [(TRIGGERED_BY Expression_Type_Pin)
   (SELECT
    [(Expression_Type_Pin = "Real_Type_Constraint) (SET_CELL Result_Type_Pin "Real_Type_Constraint))
    [(Expression_Type_Pin = "Integer_Type_Constraint) (SET_CELL Result_Type_Pin "Integer_Type_Constraint))
    [OTHERWISE (ERROR "Operand of unary plus operation must be integer or real')]]])]]

[DEFINE_CONSTRAINT And_Constraint ;the definition for Or is identical.

[PINS
  (Result (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
  (Left_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
  (Right_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))]]

```



```

[CAN_BE_ONE_OF
  (Result.Syntax_Pin Expression)
  (Result.Environment_Pin ?)
  (Result.Type_Pin "Boolean_Type_Constraint")
  (Result.Assign_Attribute_Pin FALSE)
  (Left.Expression,Syntax_Pin Expression)
  (Left.Expression,Environment_Pin ?)
  (Left.Expression,Type_Pin "Boolean_Type_Constraint")
  (Left.Expression,Assign_Attribute_Pin don't_care)
  (Right.Expression,Syntax_Pin Expression)
  (Right.Expression,Environment_Pin ?)
  (Right.Expression,Type_Pin "Boolean_Type_Constraint")
  (Right.Expression,Assign_Attribute_Pin don't_care)]
[MUST_CONNECT Result Left.Expression Right.Expression]

[INITIALIZE
  (CONSTANT_CELL Result,Syntax_Pin Expression)
  (CONSTANT_CELL Left.Expression,Syntax_Pin Expression)
  (CONSTANT_CELL Right.Expression,Syntax_Pin Expression)
  (CONSTANT_CELL Result,Type_Pin "Boolean_Type_Constraint")
  (CONSTANT_CELL Left.Expression,Type_Pin "Boolean_Type_Constraint")
  (CONSTANT_CELL Right.Expression,Type_Pin "Boolean_Type_Constraint")
  (CONSTANT_CELL Result,Assign_Attribute_Pin FALSE)]

[Environment_Rules
  ((TRIGGERED_BY Result.Environment_Pin)
   (SET_CELL Left.Expression,Environment_Pin Result.Environment_Pin)
   (SET_CELL Right.Expression,Environment_Pin Result.Environment_Pin))]]

[DEFINE_CONSTRAINT Not_Constraint

  IFIRS
    (Result (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
    (Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))

  [CAN_BE_ONE_OF
    (Result,Syntax_Pin Expression)
    (Result,Environment_Pin ?)
    (Result,Type_Pin "Boolean_Type_Constraint")
    (Result,Assign_Attribute_Pin FALSE)]

```

```

(Expression.Syntax.Pin
(Expression.Environment.Pin
(Expression.Type.Pin
(Expression.Assign_Attribute.Pin don't_care))
[MUST_CONNECT Result Expression]

[INITIALIZE
(CONSTANT_CELL Result.Syntax.Pin Expression)
(CONSTANT_CELL Expression.Syntax.Pin Expression)
(CONSTANT_CELL Result.Type.Pin "Boolean_Type_Constraint")
(CONSTANT_CELL Expression.Type.Pin "Boolean_Type_Constraint")
(CONSTANT_CELL Result.Assign_Attribute.Pin FALSE)]

[Environment_Rules
[ (TRIGGERED_BY Result.Environment.Pin)
 (SET_CELL Expression.Environment.Pin Result.Environment.Pin)]]]

[DEFINE_CONSTRAINT Less_Than_Constraint if the definitions for Less_Or_Equal, Equal, Not_Equal,
Greater_Than, and Greater_Or_Equal are identical.

[PINS
(Result
(Left.Expression (Syntax.Pin Environment.Pin Type.Pin Assign_Attribute.Pin))
(Right.Expression (Syntax.Pin Environment.Pin Type.Pin Assign_Attribute.Pin))
[CAN_BE_ONE_OF
(Result.Syntax.Pin Expression)
(Result.Environment.Pin ?)
(Result.Type.Pin "Boolean_Type_Constraint")
(Result.Assign_Attribute.Pin FALSE)
(Left.Expression.Syntax.Pin Expression)
(Left.Expression.Environment.Pin ?)
(Left.Expression.Type.Pin ("Integer_Type_Constraint "Real_Type_Constraint "Boolean_Type_Constraint"))
(Right.Expression.Syntax.Pin Expression)
(Right.Expression.Environment.Pin ?)
(Right.Expression.Type.Pin ("Integer_Type_Constraint "Real_Type_Constraint "Boolean_Type_Constraint"))
(Right.Expression.Assign_Attribute.Pin don't_care)]
[MUST_CONNECT Result Left.Expression Right.Expression]

```

```

INITIALIZE
(CONSTANT_CELL Result,Syntax_Pin Expression)
(CONSTANT_CELL Left_Expression,Syntax_Pin Expression)
(CONSTANT_CELL Right_Expression,Syntax_Pin Expression)
(CONSTANT_CELL Result,Type_Pin "Boolean_Type_Constraint")
(CONSTANT_CELL Result,Assign_Attribute_Pin FALSE)]

[Environment_Rules
[TRIGGERED_BY Result,Environment_Pin]
[SET_CELL Left_Expression,Environment_Pin Result,Environment_Pin]
[SET_CELL Right_Expression,Environment_Pin Result,Environment_Pin]]]

[Type_Rules
[TRIGGERED_BY Left_Expression]
[SELECT
[Left_Expression,Type_Pin = "Integer_Type_Constraint" OK]
[Left_Expression,Type_Pin = "Real_Type_Constraint" OK]
[Left_Expression,Type_Pin = "Boolean_Type_Constraint" (SET_CELL Right_Expression,Type_Pin "Boolean_Type_Constraint")]
[OTHERWISE (ERROR "Left operand of relational operator must be integer, real or boolean")]]]

[TRIGGERED_BY Right_Expression]
[SELECT
[Right_Expression,Type_Pin = "Integer_Type_Constraint" OK]
[Right_Expression,Type_Pin = "Real_Type_Constraint" OK]
[Right_Expression,Type_Pin = "Boolean_Type_Constraint" (SET_CELL Left_Expression,Type_Pin "Boolean_Type_Constraint")]
[OTHERWISE (ERROR "Right operand of relational operator must be integer, real or boolean")]]]

[DEFINE_CONSTRAINT Array_Selector_Constraint

[PINS
(Parent (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
(Left_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))
(Right_Expression (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))]

[CAN_BE_ONE_OF
(Parent,Syntax_Pin Expression)
(Parent,Environment_Pin ?)
(Parent,Type_Pin the element_type_of_the_array_being_selected)
(Parent,Assign_Attribute_Pin TRUE)
(Left_Expression,Syntax_Pin Expression)
(Left_Expression,Environment_Pin ?)
(Left_Expression,Type_Pin "Array_Constructor_Type_Constraint")

```

```

(Left_Expression.Assign_Attribute_Pin don't_care)
(Right_Expression.Syntax_Pin
 Expression)
(Right_Expression.Environment_Pin
 ?)
(Right_Expression.Type_Pin
 "Integer_Type_Constraint")
(Right_Expression.Assign_Attribute_Pin don't_care)]
[MUST_CONNECT Parent Left_Expression Right_Expression]

[INITIALIZE
 (CONSTANT_CELL Parent.Syntax_Pin
 Expression)
 (CONSTANT_CELL Left_Expression.Syntax_Pin
 Expression)
 (CONSTANT_CELL Right_Expression.Syntax_Pin
 Expression)
 (CONSTANT_CELL Right_Expression.Type_Pin "Integer_Type_Constraint"); Remember: indices of our arrays must be integers.
 (CONSTANT_CELL Parent.Assign_Attribute_Pin TRUE)]

[Environment_Rules
 ((TRIGGERED_BY Parent.Environment_Pin)
 (SET_CELL Left_Environment.Environment_Pin Result.Environment_Pin)
 (SET_CELL Right_Environment.Environment_Pin Result.Environment_Pin))]

[Type_Rules
 ((TRIGGERED_BY Left_Expression.Type_Pin)
 (IF (Left_Expression.Type_Pin = "Array_Constructor_Type")
 THEN (SET_CELL Parent.Type_Pin (THE_VALUE_IN_CELL Element_Type_Pin Left_Expression.Type_Pin))
 ELSE (ERROR "Attempting to index a non-array variable")))]

[DEFINE_CONSTRAINT Constant_Constraint

 [PINS
 (Parent (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin))]

 [CAN_BE_ONE_OF
 (Parent.Syntax_Pin (Constant_Expression))
 (Parent.Environment_Pin (don't_care))
 (Parent.Type_Pin ("Integer_Type_Constraint "Real_Type_Constraint "Boolean_Type_Constraint"))
 (Parent.Assign_Attribute_Pin FALSE)]

 [MUST_CONNECT Parent]

 [EXTRAS Actual_Integer_Value]

 [INITIALIZE

```

```

(CONSTANT_CELL Parent.Assign_Attribute_Pin FALSE)]

[Syntax_Pin_Rules
 [ (TRIGGERED_BY Parent.Syntax_Pin)
  (SELECT
   [ (Parent.Syntax_Pin = Constant) OK] ifor use in Lower and Upper bounds in array constructors.
   [ (Parent.Syntax_Pin = Expression) OK]
   [ OTHERWISE (ERROR "Constants can only be used in arithmetic expressions and as bounds in array declarations'') ] ] ] ] ]

[DEFINE_CONSTRAINT Usage_Identifier_Constraint
 [PINS
  (Parent (Syntax_Pin Environment_Pin Type_Pin Assign_Attribute_Pin)) ]
 [CAN_BE_ONE_OF
  (Parent.Syntax_Pin Expression)
  (Parent.Environment_Pin ?)
  (Parent.Type_Pin ("Integer_Type_Constraint "Real_Type_Constraint "Boolean_Type_Constraint "Array_Constructor_Type_Constraint "Procedure_Constraint))
  (Parent.Assign_Attribute_Pin (TRUE FALSE))
 [MUST_CONNECT Parent]
 [EXTRAS Actual_Name]
 [INITIALIZE
  (CONSTANT_CELL Parent.Syntax_Pin Expression)]
 [Environment_Rules
  [ (TRIGGERED_BY Parent.Environment_Pin)
   (LET (Declaration (LOOKUP_USAGE_IDENTIFIER Actual_Name Parent.Environment_Pin))
    (IF (NULL Declaration)
     THEN (ERROR "Undefined identifier " Actual_Name)
     ELSE
      (SET_CELL Parent.Type_Pin (TYPE_OF Declaration))
      (SET_CELL Parent.Assign_Attribute_Pin (THE_VALUE_OF_EXTRA Assignment_Attribute_Declaration)))) ] ] ] ] ]

```

Functions
For The
Simple Design

A-26

```

[DEFUN TYPE_OF (Declaration)
  (SELECT
    [(Declaration = Procedure_Constraint) ~Procedure_Constraint]
    [(Declaration = Var_Constraint) (THE_VALUE_IN_CELL Type.Pin Definition)]
    [OTHERWISE (ERROR 'Invalid usage of program name')]))]

[DEFUN LOOKUP_DEFINITION_IDENTIFIER (Name Local_Environment)
  ;look in the local environment only, for a declaration constraint
  ; that has its Name.Actual_Name.Pin equal to 'Name'.
  [FOREACH (Declaration IN (the list of local declarations contained in this environment)) DO
    (IF (Name = (THE_VALUE_IN_CELL Name.Actual_Name.Pin Declaration))
      THEN (RETURN Declaration))]
  (RETURN NIL)]

[DEFUN LOOKUP_USAGE_IDENTIFIER (Name Local_Environment)
  [FOREACH (Declaration IN (the list of local declarations contained in this environment)) DO
    (IF (Name = (THE_VALUE_IN_CELL Name.Actual_Name.Pin Declaration))
      THEN (RETURN Declaration)))]
  [LET (Local_Environment_Of_Container (THE_VALUE_IN_CELL Parent.Environment.Pin Scope_Root))
    ; If ever Local_Environment_Of_Container is NIL it is either because the
    ; user's procedure is not connected (directly or indirectly)
    ; to a Program node, OR we have reached the STANDARD_ENVIRONMENT
    ; and still cannot find a declaration for the Name.
    (IF (NULL Local_Environment_Of_Container)
      THEN (RETURN NIL)
      ELSE (LOOKUP_USAGE_IDENTIFIER Name Local_Environment_Of_Container)))]

```

Improved Design Constraints

We define here, only those constraints which differ from those of the simple design. The statement constraints (If, While, Assignment and Call) are almost identical to what they were in the simple design except that the "Parent" syntax pin and the "Parent" environment pin have been split into separate branches. The expression constraints (arithmetic operators, boolean operators, relational operators, array selections, and the constants) are the same as in the simple design.

Notice that in the improved design, a rule can have both a Propagation Part and a Retraction Part. If not specified, then the rule is a Propagation rule.


```

[DEFINE_CONSTRAINT Program_Constraint
[PINS
(Name (Environment.Pin))
(Local_Declarations (Syntax.Pin Environment.Pin))
(External_Declaration (Environment.Pin))
(Statements (Syntax.Pin Environment.Pin))]]
[[CAN_BE_ONE_OF
(Parent.Environment.Pin "STANDARD_ENVIRONMENT")
(Name.Syntax.Pin Declaration_Identifier)
(Name.Environment.Pin "STANDARD_ENVIRONMENT")
(Local_Declarations.Syntax.Pin Declaration)
(Statements.Syntax.Pin Statement)
(Statements.Environment.Pin "ME")]
[MUST_CONNECT Name_Statements]
INITIALIZE
(CONSTANT_CELL Name.Syntax.Pin Declaration_Identifier)
(CONSTANT_CELL Local_Declarations.Syntax.Pin Declaration)
(CONSTANT_CELL Statements.Syntax.Pin Statement)
(CONSTANT_CELL Parent.Environment.Pin "STANDARD_ENVIRONMENT")
(CONSTANT_CELL Name.Environment.Pin "STANDARD_ENVIRONMENT")
(CONSTANT_CELL Local_Declarations.Environment.Pin "ME")
(CONSTANT_CELL Statement.Environment.Pin "ME")]
[DEFINE_CONSTRAINT Procedure_Constraint
[PINS
(Parent.Syntax (Syntax.Pin))
(Parent.Environment (Environment.Pin))
(Name (Syntax.Pin Environment.Pin Actual_Name))
(Local_Declarations (Syntax.Pin Environment.Pin))
(Statements (Syntax.Pin Environment.Pin))
(Usages (Type.Pin Assign_Attribute_Pin))]]
[[CAN_BE_ONE_OF
(Parent.Syntax.Pin Declaration)

```

```

(Parent_Environment_Pin ?)
(Name_Syntax_Pin Declaration_Identifier)
(Name_Environment_Pin ?)
(Name_Actual_Name ?)
(Local_Declarations_Syntax_Pin Declaration)
(Local_Declarations_Environment_Pin ~ME)
(Statements_Syntax_Pin Statement)
(Statements_Environment_Pin ~ME)
(Uses_Type_Pin ~ME)
(Uses_Assign_Attribute_Pin FALSE))
[MUST_CONNECT Parent_Syntax Name Statements]

[INITIALIZE
(CONSTANT_CELL Parent_Syntax_Syntax_Pin Declaration)
(CONSTANT_CELL Name_Syntax_Pin Declaration_Identifier)
(CONSTANT_CELL Local_Declarations_Syntax_Pin Declaration)
(CONSTANT_CELL Statements_Syntax_Pin Statement)
(CONSTANT_CELL Local_Declarations_Environment_Pin ~ME)
(CONSTANT_CELL Statements_Environment_Pin ~ME)
(CONSTANT_CELL Uses_Type_Pin ~ME)
(CONSTANT_CELL Uses_Assign_Attribute_Pin FALSE))

[Environment Rules
((TRIGGERED_BY Parent_Environment_Environment_Pin)
 (SET_CELL Name_Environment_Pin Parent_Environment_Environment_Pin)
 (SET_CELL External_Declarations_Environment_Pin Parent_Environment_Environment_Pin))]
F((TRIGGERED_BY Name_Actual_Name)
 [RETRACTION
 (REMOVE_DECLARATION ~ME Name_Environment_Pin)]
 [PROPAGATION
 (INSERT_DECLARATION ~ME Name_Environment_Pin)]])

[DEFINE_CONSTRAINT Var_Constraint
 [PINS

```

```

(Parent_Syntax (Syntax_Pin))
(Parent_Environment (Environment_Pin))
(Name (Syntax_Pin Environment_Pin Actual_Name_Pin))
(Type (Syntax_Pin Type_Pin))
(Usages (Type_Pin Assign_Attribute_Pin)))

[CAN_BE_ONE_OF
(Parent_Syntax_Syntax_Pin Declaration)
(Parent_Environment_Environment_Pin ?)

(Name_Syntax_Pin Declaration_Identifier)
(Name_Environment_Pin ?)
(Name_Actual_Name ?)

(Type_Syntax_Pin Type)
(Type_Type_Pin ?)

(Usages_Type_Pin ?)
(Usages_Assign_Attribute_Pin TRUE))]

[MUST_CONNECT Parent Name Type]

[INITIALIZE
(CONSTANT_CELL Parent_Syntax_Syntax_Pin Declaration)
(CONSTANT_CELL Name_Syntax_Pin Declaration_Identifier)
(CONSTANT_CELL Type_Syntax_Pin Type)
(CONSTANT_CELL Usages_Assign_Attribute_Pin TRUE))]

[Environment_Rules
[[(TRIGGERED_BY Parent_Environment_Environment_Pin)
(SET_CELL Name_Environment_Pin Parent_Environment_Environment_Pin)]

[(TRIGGERED_BY Name_Actual_Name_Pin)
[RETRACTION
(REMOVE_DECLARATION "NE Name_Environment_Pin)]

[PROPAGATION
(INSERT_DECLARATION "NE Parent_Environment_Environment_Pin)]]]

[Type_Rules
[[(TRIGGERED_BY Type_Type_Pin)
(SET_CELL Usages_Type_Pin Type_Type_Pin)]]]

[DEFINE_CONSTRAINT Pseudo_Declaration_Constraint

```

```

[PINS
  (Parent_Environment (Environment_Pin))
  (Name (Syntax_Pin Environment_Pin Actual_Name_Pin))
  (Declaration (Type_Pin Assign_Attribute_Pin))
  (Usages (Type_Pin Assign_Attribute_Pin))
[CAN_BE_ONE_OF
  (Parent_Environment.Environment_Pin ?)
  (Name.Syntax.Pin Declaration_Identifier)
  (Name.Environment.Pin ?)
  (Name.Actual_Name.Pin ?)
  (Declaration.Type.Pin ?)
  (Declaration.Assign_Attribute.Pin ?)
  (Usages.Type.Pin ?)
  (Usages.Assign_Attribute.Pin ?)]
[MUST_CONNECT]

[INITIALIZE
  (CONSTANT_CELL Name.Syntax.Pin Declaration_Identifier)]

[Environment_Rules
  [(TRIGGERED_BY Parent_Environment.Environment_Pin)
   (SET_CELL Name.Environment.Pin Parent_Environment.Environment_Pin)]
  [(TRIGGERED_BY Name.Actual_Name_Pin)
   (LOOKUP_USAGE "ME Name.Actual_Name_Pin Parent_Environment.Environment_Pin)]]

[Type_Rules
  [(TRIGGERED_BY Declaration.Type_Pin)
   (SET_CELL Usages.Type_Pin Declaration.Type_Pin)]
  [(TRIGGERED_BY Declaration.Assign_Attribute_Pin)
   (SET_CELL Usages.Assign_Attribute_Pin Declaration.Assign_Attribute_Pin)]]

[DEFINE_CONSTRAINT Declaration_Identifier_Constraint

[PINS
  (Parent (Syntax_Pin Environment_Pin Identifier_Name_Pin))]
[CAN_BE_ONE_OF
  (Parent.Syntax.Pin Declaration_Identifier)

```

```

(Parent.Environment.Pin
(Parent.Identifier_Name.Pin ?))
[MUST_CONNECT Parent]
[EXTRAS Actual_Name]
[INITIALIZE
(CONSTANT_CELL Parent.Syntax_Pin Declaration_Identifier)]
[Environment.Pin_Rules
[(TRIGGERED_BY Parent.Environment.Pin)
(IF (LOOKUP_DEFINITION_IDENTIFIER Actual_Name Parent.Environment.Pin))
THEN (ERROR Actual_Name ' is already defined in this local environment')
ELSE (SET_CELL Parent.Identifier_Name.Pin Actual_Name))]]]

[DEFINE_CONSTRAINT Integer_Type_Constraint
[PINS
(Parent (Syntax_Pin Type_Pin))]
[CAN_BE_ONE_OF
(Parent.Syntax_Pin Type)
(Parent.Type_Pin ~ME)]
[MUST_CONNECT Parent]
[INITIALIZE
(CONSTANT_CELL Parent.Syntax_Pin Type)
(CONSTANT_CELL Parent.Type_Pin ~ME)]]

[DEFINE_CONSTRAINT Real_Type_Constraint
[PINS
(Parent (Syntax_Pin Type_Pin))]
[CAN_BE_ONE_OF
(Parent.Syntax_Pin Type)
(Parent.Type_Pin ~ME)]
[MUST_CONNECT Parent]
[INITIALIZE
(CONSTANT_CELL Parent.Syntax_Pin Type)
(CONSTANT_CELL Parent.Type_Pin Type)]]

```

```

(CONSTANT_CELL Parent.Type_Pin ~ME)]]

[DEFINE_CONSTRAINT Boolean_Type_Constraint
[PINS
(Parent (Syntax_Pin Type_Pin))]
[CAN_BE_ONE_OF
(Parent.Syntax_Pin Type)
(Parent.Type_Pin ~ME)]
[MUST_CONNECT Parent]

[INITIALIZE
(CONSTANT_CELL Parent.Syntax_Pin Type)
(CONSTANT_CELL Parent.Type_Pin ~ME)]]

[DEFINE_CONSTRAINT Array_Constructor_Type_Constraint
[PINS
(Parent (Syntax_Pin Type_Pin))
(Lower_Bound (Syntax_Pin Environment.Pin Type_Pin Assign_Attribute_Pin))
(Upper_Bound (Syntax_Pin Environment.Pin Type_Pin Assign_Attribute_Pin))
(Element_Type (Syntax_Pin Type_Pin))]
[CAN_BE_ONE_OF
(Parent.Syntax_Pin Type)
(Parent.Type_Pin ~ME)
(Lower_Bound.Syntax_Pin Constant)
(Lower_Bound.Environment.Pin don't_care)
(Lower_Bound.Type_Pin ~Integer_Type_Constraint)
(Lower_Bound.Assign_Attribute_Pin don't_care)
(Upper_Bound.Syntax_Pin Constant)
(Upper_Bound.Environment.Pin don't_care)
(Upper_Bound.Type_Pin ~Integer_Type_Constraint)
(Upper_Bound.Assign_Attribute_Pin don't_care)
(Element_Type.Syntax_Pin Type)
(Element_Type.Type_Pin ?)]
[MUST_CONNECT Parent Lower_Bound Upper_Bound Element_Type]

```

POOR PRINT
Epreuve illisible

```

[INITIALIZE
  (CONSTANT_CELL Parent.Syntax_Pin Type)
  (CONSTANT_CELL Lower_Bound.Syntax_Pin Constant)
  (CONSTANT_CELL Upper_Bound.Syntax_Pin Constant)
  (CONSTANT_CELL Element_Type.Syntax_Pin Type)
  (CONSTANT_CELL Lower_Bound.Type_Pin Integer_Type)
  (CONSTANT_CELL Upper_Bound.Type_Pin Integer_Type)]
[Type_Rules
  ((TRIGGERED_BY Element_Type.Type_Pin)
   (SET_CELL Parent.Type_Pin "ME"))]
[DEFINE_CONSTRAINT Usage_Identifier_Constraint
  IPINS
  (Parent
   (Declaration (Type_Pin Assign_Attribute_Pin)
    Unbound_Or_External_Pin)
   (CAN_BE_ONE_OF
    (Parent.Syntax_Pin Expression)
    (Parent.Environment_Pin ?)
    (Parent.Type_Pin ?)
    (Parent.Assign_Attribute_Pin ?)
    (Declaration.Type_Pin ?)
    (Declaration.Assign_Attribute_Pin ?)
    (Unbound_Or_External_Pin NIL))
   (MUST_CONNECT Parent)
   (CYTRAS Actual_Name Environment_Of_Declaration)]
[INITIALIZE
  (CONSTANT_CELL Parent.Syntax_Pin Expression)]
[Environment_Rules
  ((TRIGGERED_BY Parent.Environment_Pin)
   (LOOKUP_USAGE "ME Actual_Name Parent.Environment_Pin"))]
[Type_Rules
  ((TRIGGERED_BY Declaration.Type_Pin)
   (SET_CELL Parent.Type_Pin Declaration.Type_Pin))]

```

[Assign_Attribute_Rules
[(/TRIGGERED_BY Declaration.Assign_Attribute_Pin)
(SET_CELL Parent.Assign_Attribute_Pin Declaration.Assign_Attribute_Pin)]]

AR

A-36

Functions
For The
Improved Design

```

CDEFUN MAKE_LIST (Pin)
  (RETURN a list of all the other cells which are in the same node as the cell attached to 'Pin'))

CDEFUN LOOKUP_DEFINITION_IDENTIFIER (Name Local_Environment)
  (FOREACH (Declaration IN (the list of local declarations contained in this environment)) DO
    (IF (Name = (THE_VALUE_IN_CELL Name.Actual_Name.Pin OF Declaration))
      THEN (RETURN Declaration)))
  (RETURN NIL))

CDEFUN LOOKUP_USAGE Usage Name Local_Environment
  (FOREACH Local_Declaration IN (the list of local declarations in the Local_Environment) DO
    (IF (Name = (THE_VALUE_IN_CELL Name.Actual_Name.Pin OF Local_Declaration))
      THEN
        (BIND Usage Local_Declaration)
        (RETURN)))
  ;Name is not declared locally
  (FOREACH Pseudo_Declaration IN (the list of external declarations in the Local_Environment) DO
    (IF (Name = (THE_VALUE_IN_CELL Name.Actual_Name.Pin OF Pseudo_Declaration))
      THEN
        (BIND Usage Pseudo_Declaration)
        (RETURN)))
  ;Name does not exist in the local environment at all!
  (CREATE_PSEUDO_DECLARATION New_Pseudo_Declaration Name)
  (INSERT New_Pseudo_Declaration IN (list of external declarations in the Local_Environment)))

CDEFUN BIND Usage Declaration
  (CONNECT (THE Declaration.Type.Pin OF Usage) (THE Usages.Type.Pin OF Declaration))
  (CONNECT (THE Declaration.Assign_Attribute.Pin OF Usage) (THE Usages.Assign_Attribute.Pin OF Declaration)))

CDEFUN REMOVE_DECLARATION Local_Declaration Local_Environment
  (IF (NOT_EMPTY MAKE_LIST (THE Usages OF Local_Declaration)))
    THEN
      ;Create a pseudo declaration which will inherit all of the
      ;usages of the local declaration that we are about to delete.
      (CREATE_PSEUDO_DECLARATION New_Pseudo_Declaration (THE Name.Actual_Name.Pin OF Local_Declaration))
      (CONNECT (THE Usages.Type.Pin OF New_Pseudo_Declaration)
        (THE Usages.Type.Pin OF Local_Declaration))
      (CONNECT (THE Usages.Assign_Attribute.Pin OF New_Pseudo_Declaration)
        (THE Usages.Assign_Attribute.Pin OF Local_Declaration))

```

```

(DISCONNECT (THE Uses.Type.Pin OF Local_Declaration))
(DISCONNECT (THE Uses.Assign.Attribute.Pin OF Local_Declaration))
(CONNECT (THE Parent_Environment.Environment.Pin OF New_Pseudo_Declaration)
 (THE External_Declarations.Environment.Pin OF Local_Environment)))]]

[DEFIN INSERT_DECLARATION New_Local_Declaration Local_Environment
 [FOREACH Pseudo_Declaration IN (the list of external declarations in the Local_Environment) DO
  (IF ((THE Name.Actual_Name.Pin OF New_Local_Declaration) = (THE Name.Actual_Name.Pin OF Pseudo_Declaration))
   THEN
    ;the new local declaration should inherit all of the
    ;uses of the pseudo declaration which we are about to
    ;remove.
    (DISCONNECT (THE Declaration.Type.Pin OF Pseudo_Declaration))
    (DISCONNECT (THE Declaration.Assign.Attribute.Pin OF Pseudo_Declaration))
    (CONNECT (THE Uses.Type.Pin OF New_Local_Declaration)
     (THE Uses.Type.Pin OF Pseudo_Declaration))
    (CONNECT (THE Uses.Assign.Attribute.Pin OF New_Local_Declaration)
     (THE Uses.Assign.Attribute.Pin OF Pseudo_Declaration))
    (DISCONNECT (THE Parent_Environment.Environment.Pin OF Pseudo_Declaration))
    RETURN)]])

```

BIBLIOGRAPHY

- [Alberga, Brown, Leeman, Mikelsons, and Wegman 79]
Alberga, C. N., Brown, A. L., Leeman, G. B. Jr., Mikelsons, M., Wegman, M. N., "A Program Development Tool", I.B.M. Research Report, June 1979.
- [Allen 79]
Allen, John R., "Don't Overlook Lisp", Byte Magazine, March 1979. PP 6-216.
- [Balzer 81]
Balzer, R., "Transformational Implementation: An Example". IEEE Transactions on Software Engineering, Vol. SE-7, pp 3-14, January 1981.
- [Barstow 79]
Barstow, D. R., "Knowledge Based Program Construction". New York: Elsevier, 1979.
- [Bobrow 72]
Bobrow, Daniel G. "Requirements for Programming Systems for List Processing". Communications of the A.C.M. Volume 15, Number 7, July 1972. PP 618-627.
- [Cheatham 81] Cheatham, T. E., "An Overview of the Harvard Program Development System", in Software Engineering Environments, H. Hunke, Ed. Amsterdam, The Netherlands, North-Holland, pp 253-266, 1981.
- [Cicarelli 78]
Cicarelli, Eugene, "An Introduction to the EMACS Editor" MIT AI Memo 447, Cambridge, January 1978.
- [Collins, Johnson, Ruttenburg, and Shapiro 80]
Collins, E., Johnson, L., Ruttenburg, J., and Shapiro, E., "PASES: A Programming Environment for PASCAL" 1980.
- [Demers, Reps, and Teitelbaum 83]
Demers, Alan, Reps, Thomas, and Teitelbaum, Tim, "Incremental Context-Dependent Analysis for Language-Based Editors", ACM Transactions on Programming Languages and Systems, July 1983. PP 449-477.
- [des Rivieres 80]
des Rivieres, James J., "The Design of an Interactive Program Manipulation System", M.Sc. thesis, Department of Computer Science, University

BIBLIOGRAPHY (cont)

of Toronto, May 1980.

[Donzeau-Gouge, Huet, Kahn and Lang 80]

Donzeau Gouge V., Huet G., Kahn G., Lang B.,
"Programming Environments based on Structured
Editors: The Mentor Experience" Institut National de
Recherche en Informatique et en Automatique,
Roquencourt, France, July 1980.

[Doyle 78]

Doyle, Jon. "Truth Maintenance Systems For Problem
Solving". Tech. Report AI-TR-419. M.I.T. Artificial
Intelligence Laboratory. Cambridge, January 1978.

[Feiler and Medina-Mora 81]

Feiler, Peter H. and Medina-Mora, Raul, "An
Incremental Programming Environment", Department of
Computer Science, Pittsburgh, Proceedings: 5'th
International Conference on Software Engineering,
San Diego, California, March 1981.

[Fraser 81]

Fraser, Christopher W. "Syntax-Directed Editing of
General Data Structures". Proceedings of the A.C.M.
Sigplan Sigoa Symposium on Text Manipulation.
Portland, Oregon, June 1981. A.C.M. Sigplan
Notices, Volume 16, Number 6, June 1981. PP 17-21.

[Fraser and Lopez 81]

Fraser Christopher W. and Lopez, A. A., "Editing
Data Structures". A.C.M. Transactions on
Programming Languages and Systems, April 1981.

[Good 81]

Good, Michael. "Etude and the Folklore of User
Interface Design". Proceedings of the A.C.M.
Sigplan Sigoa Symposium on Text Manipulation.
Portland, Oregon, June 1981. A.C.M. Sigplan
Notices, Volume 16, Number 6, June 1981. PP 34-43.

[Kant and Barstow 81] Kant, Elaine and Barstow, David R., "
The Refinement Paradigm: The Interaction of Coding
and Efficiency Knowledge in Program Synthesis".
IEEE Transaction on Software Engineering, Vol. SE-7,
No. 5, pp. 458-471, September 1981.

[Lewis 81]

Lewis J. W., "Beyond ALBE/P: Language Neutral Form",
General Electric, Corporate Research and Development

BIBLIOGRAPHY (cont)

Center, Proceedings: 5'th International Conference on Software Engineering, San Diego, California, March 1981. PP 422-429.

[McAllester 78]

McAllester, David A. "A Three Valued Truth Maintenance System". A. I. Memo 473. M.I.T. Artificial Intelligence Laboratory. Cambridge, May 1978.

[Morris Schwartz 81]

Morris, Joseph M., Schwartz, Mayer D. "The Design of a Language-Directed Editor for Block-Structured Languages". Proceedings of the A.C.M. Sigplan Sigma Symposium on Text Manipulation. Portland, Oregon, June 1981. A.C.M. Sigplan Notices, Volume 16, Number 6, June 1981. PP 28-33.

[Myers 80]

Myers, Brad A., "Displaying Data Structures for Interactive Debugging", XEROX Palo Alto Research Center, June 1980.

[Ravenel 79]

Ravenel, Bruce W. "Specification for Computer Programming Language Pascal". COMPUTER magazine, April 1979. PP 69-82.

[Stallman 79]

Stallman, Richard M., "EMACS: The Extensible, Customizable, Self-Documenting Display Editor", MIT AI Memo 519, Cambridge, June 1979.

[Stallman 80]

Stallman, Richard M. "EMACS Manual for TWENEX Users", MIT AI Memo 555, Cambridge, September 1980.

[Steele 80]

Steele, Guy Lewis Jr. "The Definition and Implementation of a Computer Programming Language Based On Constraints". Tech. Report AI-TR-595. M.I.T. Artificial Intelligence Laboratory. Cambridge, August 1980.

[Steele Sussman 79]

Steele, Guy Lewis Jr., Sussman, Gerald Jay. "Constraints". AI Memo 502. M.I.T. Artificial Intelligence Laboratory (Cambridge, November 1978). Invited paper. Proceedings APL '79 ACM SIGPLAN STAPL

BIBLIOGRAPHY (cont)

APL, Quote Quad 9, 4 (June 1979), PP 208-225.

[Teitelbaum 79]

Teitelbaum, Tim, "The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS", Tech. Report No. TR79-370, Department of Computer Science, Cornell University, June 1979.

[Teitelbaum Reps Horwitz 81]

Teitelbaum, Tim, Reps, Thomas, Horwitz, Susan. "The Why and Wherefore of the Cornell Program Synthesizer" Proceedings of the A.C.M. Sigplan Sigoa Symposium on Text Manipulation. Portland, Oregon, June 1981. A.C.M. Sigplan Notices, Volume 16, Number 6, June 1981. PP 8-16.

[Teitelman 78]

Teitelman, W., "INTERLISP Reference Manual", Xerox Palo Alto Research Center, 1974 (Revised October 1978).

[Tesler 81]

Tesler, Larry. "The Smalltalk Environment". BYTE magazine, Volume 6, Number 8, August 1981. PP 90-146.

[Waters 82]

Waters, Richard C., "The Programmer's Apprentice: Knowledge Based Program Editing". IEEE Transactions on Software Engineering, Vol SE-8, No. 1, pp 1-12, January 1982.

[Wilander 79]

Wilander, Jerker, "An Interactive Programming System for PASCAL" Research Report of Datalogi Linkoping, July 1979.

[Wile 80]

Wile, David S., "POPART: Producer of Parsers and Related Tools, System Builders' Manual", Information Sciences Institute / U.S.C., June 1980.

[Winograd 79]

Winograd, T., "Beyond Programming Languages" Communications of the ACM, July 1979. PP 391-401.

[Wood 81]

Wood, Steven R. "Z - The 95% Program Editor". Proceedings of the A.C.M. Sigplan Sigoa Symposium on

BIBLIOGRAPHY (cont)

Text Manipulation. Portland, Oregon, June 1981.
A.C.M. Sigplan Notices, Volume 16, Number 6, June
1981. PP 1-7.

[Yonke 75]

Yonke, Martin D., "A Knowledgeable,
Language-Independent System for Program Construction
and Modification", Information Sciences Institute,
University of Southern California, October 1975.

END

0	6	H	0	9	H	8	4
---	---	---	---	---	---	---	---

FIN