AN ABSTRACT OF THE THESIS OF

Abbas Birjandi for the degree of Doctor of Philosophy in Computer Science presented on May 30, 1986.

Title: A Rule Based Approach To Program Development

Abstract approved: Redacted for Privacy

Theodore G. Lewis

A rule based transformational model for program development and a meta-tool based on the above model is presented. The meta-tool can be instantiated to create various program development tools such as tools for building reusable software components, language directed editors, language to language translators, program instrumentation, structured document generator, and adaptive language based prettyprinters.

This new rule based approach has two important features: 1) it is language independent and can be applied to various languages, and 2) provides a powerful escape mechanism for extending the semantics of the rules.

Instances of the meta-tool for restructuring source programs for building abstract components and their refinement to concrete instances, source-to-source translation, and source contraction and expansion tools for improving readability and understanding are described.

A Rule Based Approach To Program Development

By

Abbas Birjandi

A Thesis submited to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

Completed May 30, 1986

Commencement June 1987

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis presented May 30, 1986

DEDICATION

This dissertation is dedicated to
Dr. Bruce D. Shriver
for whom I will always be a student and friend.

ACKNOWLEDGEMENTS

I am in debt to Ted Lewis for his help, guidance, and friendship throughout this work. I have benefited greatly from his rigorous approach to the issues.

I thank my very good friend Hussain Poor Rostam for all his support, financial and moral, during all these years. He was there anytime I needed a friend and a worrier.

I thank my friends Molly Lewis and Dr. Ibrahim Eissa for all the encouragement and support in moments of dispare.

I thank Thomas M. Kinney for his friendship, dedication, and effort in making this project a success.

I thank John Kent and Vuong Cao Nguyen for all their help.

Table of Contents

List of Figures

List of Tables

A Rule Based Approach To Program Development

Chapter 1

Introduction

The Software Life Cycle can be viewed as a collection of phases through which projects pass[Sta83a]. These phases are: problem definition, requirement analysis, system design specification, detailed design specification, coding, testing, system integration, and maintenance. A number of software life cycle tools have been proposed to assist in automating one or more of the phases in the Software Life Cycle. For example, a language-directed editor is a programming environment tool which helps a programmer write syntactically correct source code during the coding phase.

These tools are typically part of a system called a *programming environment.* Programming Environments are usually programming language dependent. For example Pascal in PECAN [Rei85] and MENTOR [DVHK80], and Lisp in Interlisp [TM81].

Programming environment tools increase programmer productivity, decrease development time, reduce maintenance costs, and minimize errors. During the past few years there has been a growing number of programming environments built around various programming languages. One issue raised in favor of these programming languages and their environments has been their suitability for generating reusable software components [Deu83,Weg83], and therefore acknowledging the

importance of reusability for curtailing the high cost of software development.

Programming language features such as support for object oriented programming, parameterized programming techniques, generic packages, extensible languages, and program development based on transformation techniques offer the capability of developing reusable software components. These programming language features have gained widespread acceptance, but the drawbacks are: implementation difficulties [Hoa81], lack of desired performance [Deu83] due to the complexity of the language and its support environment, and lack of support to facilitate the reuse of already existing software components.

Program transformation is an approach which is closely related to the work reported here. A program is transformed by a tool which replaces a section of a source program with a new section of program text that performs a different function. A typical transformation has three parts: 1) a pattern which when matched against the program determines where to apply the transformation, 2) a set of conditions which further restricts where the transformation can be applied, and 3) an action procedure which creates the new program section [RW83].

1.1 Programming Language Approach

Much attention has been given to language constructs that facilitate the reuse of code. Part of this effort includes the introduction of generic packages, parameterized programs or modules, and object oriented programming. Ada, and Smalltalk-80 are examples of languages that provide features to support reusability.

Parameterized Programming

The basic idea of parameterized programming is to construct new program modules from old ones by instantiating one or more parameters [Gog83]. Then correct instantiation of the formal parameters of a module is equivalent to placing that

module into an environment in which it is guaranteed to function properly. A parameterized program unit captures the algorithmic logic of a program module independently from the data types. This permits the reuse of a sorting algorithm, for example, with integer, real, or string data types. A parameterized unit can be instantiated as three different units; one for each of the three different data types, without rewriting the logic of sorting. For an instantiation of parameterized module to work correctly parameterized programming requires specifying the interface properties that must be satisfied.

Generic packages in Ada supports parameterized programming. However Ada's generic package lacks the capability of specifying interface properties that must be satisfied [Gog83]. This short commings is a potential problem if correctness of an instance of a parameterized module depends on certain requirements being satisfied by the environment in which the module is used.

Reusability and Classes

In Smalltalk-80 the fundamental unit of organization is the *class* , which consists of generalized data object and methods (named procedures) that have access to the data. Every object described in a program is an instance of some class. It is suggested [Deu83] that Smalltalk-80 supports reuse because: 1) it encourages use of abstraction by: a) restricting the ability to refer to the implementation detail by the implementor, and b) by providing abstract and concrete data types arranged in a hierarchy which can be used for specialization or extension; 2) reuse of collection of abstract classes and their associated algorithms as a framework into which a particular application can insert their own specialized code by constructing concrete subclasses that work together; and 3) support for system reusability across variant hardware because Smalltalk-80 is based on an ideal virtual machine and retargetting the Smalltalk-80 system boils down to recognizing the VM on the target host.

The main drawback of Smalltalk-80 is lack of efficient implementation due to the *late-binding* philosophy of the Smalltalk-80 language [Deu83]. Also the drastic departure from conventional programming techniques (and notation), lack of support for reuse of existing software, and cost of adaptation although not proven are issues that one should consider.

1.2 Program Transformation Approach

Program transformation is a method of program construction by successive applications of transformation rules. Usually this process starts with a (formal) specification, that is, a formal statement of a problem or its solution, and ends with an executable program. Nearly all transformational systems are interactive; even the "fully automatic" ones require an initial user input and rely interactively on the user to resolve unexpected events [PS83].

Although many simple transformations are basically macros which specify how to implement particular high level constructs, it is worth noting that other transformations can be much less restricted in the way they operate. Such transformations are not intended to be applied only when explicitly requested by the user. Rather, they are intended to be used whenever they become applicable for any reason.

PDS [Che83] is an integrated programming support environment that has three major components: a software database, a user interface, and a collection of tools that can be called via the user interface to manipulate the software modules stored in the software database. PDS adapts EL1 as its base language. EL1 is an extensible language. EL1 is extended to provide notations for various high level constructs. Abstract programs are developed in EL1 for further refinement. The refinment to a concrete program that actually be executed is done using two mechanisms:*definition* and *transformation.* Definition is simply providing a binding (or value) for a procedure, type, data objects, etc. Transformation is replacing some *high-level* construct

by a (more) concrete construct that realizes the intended function.

One problem with existing transformational environments is that they are very much programming language dependent. This limits the portability of components represented as transformations, and limits the way transformations are stated by requiring that every intermediate state of a program being transformed fit into the syntax of the programming language. As a result, automatic translation of an algorithm written in one language into the same algorithm written in another language can not be done.

1.3 Objective Of This Work

This thesis describes a *rule-based meta-tool* which uses the transformational approach. A meta-tool is a tool used to generate other tools. This particular rule-based meta-tool removes the programming language dependency existing in current transformational systems, provides a greater degree of portability , and can be applied to a wide spectrum of transformations.

The advantage of this approach are 1) existing software can be reused , thus reducing the cost of program development, and 2) eliminating the complexity, time, and effort of creating new programming languages or programming environments to support the notion of reusability. Furthermore, since the functionality of the tools generated by the instantiation of the meta-tool are defined through rules, adaptation to changes is mostly a matter of redefining rules–this in turn reduces the cost and time of software maintenance.

Thesis

In summary, we claim that rule-based meta-tools for transforming source code offer an important alternative to building software life cycle productivity tools for the following reasons:

1. A meta-tool provides a general framework, or shell, for quickly constructing a tool from existing parts, thus, instances of the meta-tool can be obtained with a minimum of programmer effort,

2. A rule-based system enhances productivity by simplifying the job of a tool designer, and provides flexibility in the resulting tool by allowing changes in the rules, on the fly, as the tool is being used,

3. The transformational approach is intrinsically more powerful than language-dependent abstraction mechanism because transformations work across languages, restructure source code prior to binding (compiling), and lend themselves to global operations on complete systems of software.

Note however, compiler optimizers working on a common intermediate language although are part of the transformational folklore are not the focus of this work and therefore not discussed.

In the pages that follow, we give a *proof by example* of each of these significant points listed above. First, the value of a meta-tool shell is shown by implementing three tools: 1) a Generalizer/Refiner tool for manipulating reusable components, 2) a Modula-2-to-C converter for transforming algorithms (reusable design), and 3) an understandability tool for browsing source code. Second, the value of a rule-based approach is shown by the ease of writing rules to generate Modula-2 source programs from a parse tree, automatic generalization of Modula-2 source code through dynamically created rules, and restructuring of source code by application of rules. Finally, the transformational approach is validated by the wide diversity of applications (from language conversion to language directed editing), and by noting that the transformational approach is more powerful than the language abstraction approach.

A Guide To The Thesis

This thesis is presented as three papers. The first paper: *Yashar: A Rule Based Meta-Tool For Program Development* provides the foundation for the remaining work. It defines the syntax and semantics of rules, the internal data structure representing the input to an instance of the tool and provides some examples of its applications. The second paper: *Arash:A Re-Structuring Tool For Building Software Systems From Reusable Components* explains an instance of the meta-tool through which program fragments are abstracted and later refined for creation of concrete instances. The third paper:*Artimis:A Module Indexing and Source Program Reading And Understanding Environment* is a third instance of the *meta-tool* that explains the readability and understandability portions of Artimis.

1.4 References

[Che83] T.E. Cheatham. Reusability Through Program Transformations. In *Proceedings of Workshop on Reusability in Programming*, pages 122–128, The Media Works, Inc., Newport, RI, September 1983.

[Deu83] L. Peter Deutsch. Reusability In The Smalltalk-80 Programming System. In *Proceedings of Workshop on Reusability in Programming*, pages 72–76, The Media Works, Inc., Newport, RI, September 1983.

[DVHK80] V. Donzeau-Gouge, Veronique, Huet, and G. Kahn. Programming Environment Based On Structured Editors:The Mentor Experience. In *Workshop on Programming Environments*, Ridgefield, CT, June 1980.

[Gog83] Joseph Goguen. Parameterized Programming. In *Proceedings of Workshop on Reusability in Programming*, pages 138–150, The Media Works, Inc., Newport, RI, September 1983.

[Hoa81] C. A. R. Hoare. The Emperor's Old Clothes. *Communication of ACM*, 24(2), February 81.

[PS83] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, 15:199–236, September 1983.

[Rei85] Steven P. Reiss. Program Development Systems That Supports Multiple Views. *IEEE Transaction on Software Engineering*, SE-11(3), March 1985.

[RW83] Charles Rich and Richard C. Waters. Formalizing Reusable Software Components. In *Proceedings of Workshop on Reusability in Programming*, pages 152–159, The Media Works, Inc., Newport, RI, September 1983.

[Sta83] American National Standard. *IEEE Standard Glossary of Software Engineering Terminology.* New York, February 1983.

[TM81] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. *IEEE Computer Magazine*, 14(4):25–33, 1981.

[Weg83] Peter Wegner. Varieties Of Reusability. In *Proceedings of Workshop on Reusability in Programming*, pages 30–44, The Media Works, Inc., Newport,RI, 9 1983.

Chapter 2

Yashar: A Rule Based Meta-Tool For Program Development

Yashar: A Rule Based Meta-Tool For Program Development

Abbas Birjandi

T.G. Lewis

Department of Computer Science

Oregon State University

Corvallis, Oregon 97331

(503) 754-3273

Abstract:

Yashar is a generalized meta-tool which can be tailored to a wide variety of application-specific tools by hand-crafting a small number of user interface and support routines and writing a small set of rules which define transformations on input. The rule-based approach to constructing programmer's tools is a new approach which appears to have great value for extending a given tool without a large amount of additional effort; is useful in writing program restructuring tools such as translators and reusablility transformers; and is useful in writing a variety of tools that directly operate on the source code of a program, such as editors and document generators. In this paper we describe Yashar's rule processor and rule syntax and then give an example of a tool that automatically translates Modula-2 source programs into equivalent C source programs.

Keywords: Programming environment, program transformation, source code mutation, language-directed tools, rapid prototyping, source language to source language translation.

2.1 Introduction

A software tool is a generally useful program for helping with day-to-day programming tasks [KM81]. For example, a syntax-directed editor helps a programmer write syntactically correct source code. The main purpose of a tool is to increase programmer productivity, decrease development time, reduce maintenance costs, and minimize errors.

Two distinct approaches have been taken in building tools: 1) tool box system, and 2) integrated approach. In a *tool box system* the collection of tools, their application and the output produced by the tools must be directly managed by the programmer [Ost81]. For example, the *MAKE* utility [Fel79] is useful for managing the compile-link cycle.

The *integrated approach* attempts to directly automate program development by embedding tools in a high level language. The Interlisp programming environment can be considered an example of the integrated approach [TM81]. Interpreters, syntax directed editors, consistency checkers, correctness verifiers, and compilers are other examples.

A tool produced by Yashar takes advantage of both approaches, but Yashar is oriented more toward the integrated approach than the tool box approach. Like Interlisp [TM81], in which tools operate on a common representation of data (lists), Yashar tools operate on a common tree-structured representation of data. However, unlike Interlisp, a tool based on Yashar is not bound to one specific language.

Yashar is called a *meta-tool* because it can be tailored into a specific tool through modification of its operation—a subject to be described more fully in this paper. Each time Yashar is specialized to perform a certain tool function, we say the resulting tool is an *instance* of the meta-tool. It is our intention to show how a meta-tool such as Yashar can benefit both tool developers and software developers, alike.

The notion of specifying *rules* rather than writing a program each time a new programming tool is needed is the main significance of Yashar. While we have not succeeded in completely eliminating the need to write programs to build a tool, we have taken the first step toward a generalized meta-tool with Yashar. A designer might think of Yashar as a tool *shell* consisting of user interface, rules, and a rule processor for carrying out the transformations specified in the rules. We use the terms shell and meta-tool somewhat loosely, here, and often use the terms interchangeably to describe Yashar.

One might question the validity of calling an instance of Yashar a *rule-based* system. Rules can be viewed as a formalism for defining knowledge independent of the method of computation. The rules in Yashar are declarative in the sense that there is no sequencing implied by the order in which the rules appear. Each rule defines a transformation to be performed without specifying the order of performance. On the other hand, Yashar rules differ from the declarative rules used in logic programming [CM84] where a rule states a proposition corresponding to a logical implication [Col85]. Yashar rules include imperative commands which operate directly on inputs, much like the operations in Lisp which operate directly on input lists. Yashar rules are interpreted by a generalized rule processor which transforms tree-structured inputs into useful outputs. The usefulness of this approach is the central theme of this paper.

2.1.1 Objectives of Yashar

The primary goal of Yashar is to study the practicality of a rule-based meta-tool as a basis for building specialized programming tools such as:

- A tool for building reusable software components,
- An adaptive language based prettyprinter,

- A structured document generator, and
- A language-to-language translator.

Yashar has been used to build a 1) a Modula-2 to C translator which converts algorithms written in Modula-2 to their C equivalent, 2) a re-structuring tool called Arash [BL86a] for building software systems from reusable components, therefore promoting reuse of existing software systems, 3) an adaptive prettyprinter for Modula-2, and 4) portions of a program reading, understanding and indexing tool called Artimis [BL86b].

2.2 Related Work

The notion of unparsing in structured editors is the basis of Yashar. The syntax of Yashar's rules are adapted from [Fri83] which in turn has been borrowed from ALOE of GANDALF [NH81]. However the function of Yashar extends beyond a prettyprinter because semantically its extensions enable the rules to be more powerful. For example, any arbitrary computations can be defined through Yashar's escape mechanism. Yashar is more of a tree processor similar to MENTOL tree processing virtual machine in MENTOR [DVHK80]. However we could not verify if MENTOL's instruction set provides notions similar to Yashar's, such as escape mechanisms, and the ability to communicate control data [DVHK80,DKLM84].

2.3 Overall System Architecture

There are two classes of users of Yashar; 1) designers, who build application-specific instances of Yashar, and 2) programmers who use instances of Yashar during their daily programming. Designers must write routines to perform the following functions shown in Figure 2.1:

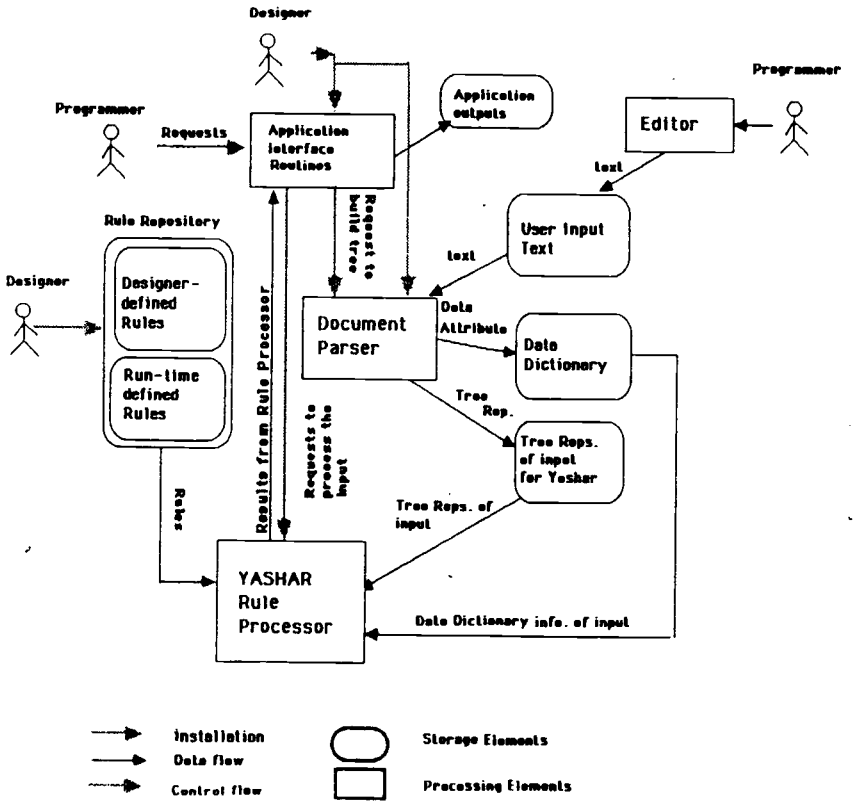


Figure 2.1: The Yashar Programming Environment

- User Interface Routines
- Document Parser

In addition, a designer must write a series of rules which are stored in a *rule repository*, see Figure 2.1. There are two kinds of rules stored in the rule repository: 1) designer-defined rules, and 2) run-time defined rules. All designer-defined rules are written by the designer and installed in the rule repository by hand. All run-time defined rules are generated automatically by User Interface Routines, which are written by the designer. These rules are destined to be used by the Rule Processor, which takes rules one at a time from the Rule Repository and uses them to transform the tree representation of the input. Figure 2.1 shows the parts of Yashar which must be installed manually by the tool designer.

Once a tool has been created, a programmer uses the tool as follows. Input text is read from a User Input Text file and converted by the Document Parser into a tree structure, and if appropriate, into Data Dictionary information. The programmer controls this process through an interface specified by the User Interface Routines. Recall that these routines are specific to a tool. A collection of User Interface Routines for a Modula-2 to C translator tool, for example, will differ from the User Interface Routines for a tool that restructures reusable modules.

As an example consider a tool for restructuring algorithms written in Modula-2 so that they can be reused in a C program. The Modula-2 to C translator which

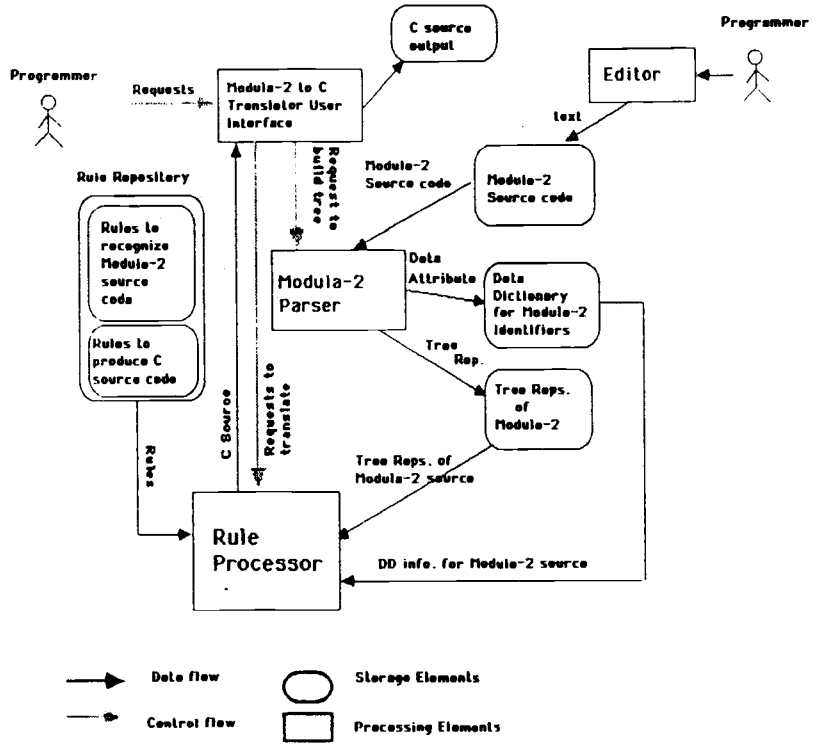


Figure 2.2: A Modula-2 to C Translator Tool

was built using Yashar is shown in Figure 2.2. The components of Figure 2.2 are application-specific versions of the components shown in Figure 2.1. In Figure 2.2 a designer has written User Interface Routines to perform the Modula-2 to C Transla-

tor User Interface functions; written a Document Parser called the Modula-2 Parser to parse Modula-2 source code and store it as an internal tree; and finally, supplied designer-defined rules for re-writing Modula-2 statements as equivalent C statements.

The input to Figure 2.2 is a Modula-2 source program file and the output is an equivalent C source program file.

It is important to notice that Modula-2 has constructs that are not supported in C. Therefore a library of support routines is needed to implement such constructs. For example, the Modula-2 TRANSFER construct has no counterpart in C and so there must be a library support routine which is functionally equivalent to TRANSFER.

2.4 User Interface Routines

User Interface Routines written by a designer are generally responsible for:

- Communicating with the user of the tool,
- Activating the rule processor,
- Capturing the results of the operation, and
- Generating new rules automatically

For example, the Modula-2 to C Translator User Interface takes care of requests to translate, selects the Modula-2 source code to be translated, and activates the Modula-2 Parser, and the rule processor respectively. Also it provides a multiple window editor through which the translated module is displayed and can be further edited or saved.

In Figure 2.3 by selecting the *Modula-2 To C* menu item, the Modula-2 To C User Interface Routine (MCUIR) displays a dialog for selection of the Modula-2 source

program for translation, see Figure 2.4. Once a Modula-2 source file is selected, MCUIR activates the Modula-2 Parser to build the internal tree representation of the selected module along with its data dictionary. Afterwards, if the parsing was successful, the rule processor is activated and the tree representation of the module along with its data dictionary is made available to the rule processor. The rule processor loads the designer-defined re-writing rules for translating Modula-2 source programs to C and performs the task of translation. Once the translation is done, the result is returned to the MCUIR. MCUIR in return displays the Modula-2 source and its translation in two separate windows for further inspection or possible modification. Figure 2.5 and 2.6 are examples of two Modula-2 programs and their

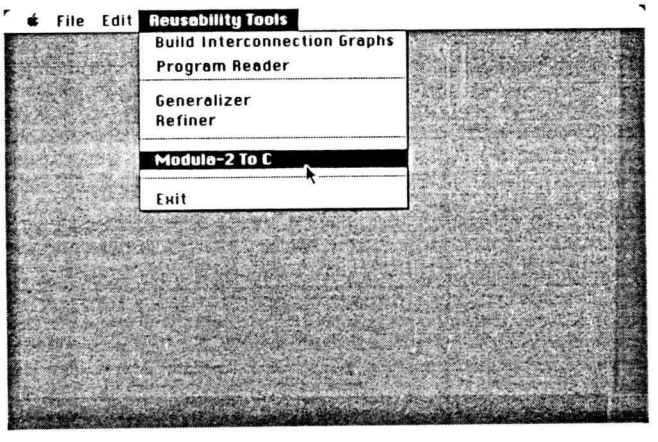


Figure 2.3: Modula-2 to C Translator User Interface

C equivalents produced by this instance of Yashar. Figure 2.5 shows a case in which there is no need for designer-defined function and translation is completely done by the rule processor without any external help. However, Figure 2.6 shows a case in which a designer-defined function is called to resolve the scope problem stemming from non-local variables referenced in nested procedures. Notice that the function for resolving the scope problem was installed in the *Function Table*, refer to section 2.6, by the MCUIR before activating the rule processor.

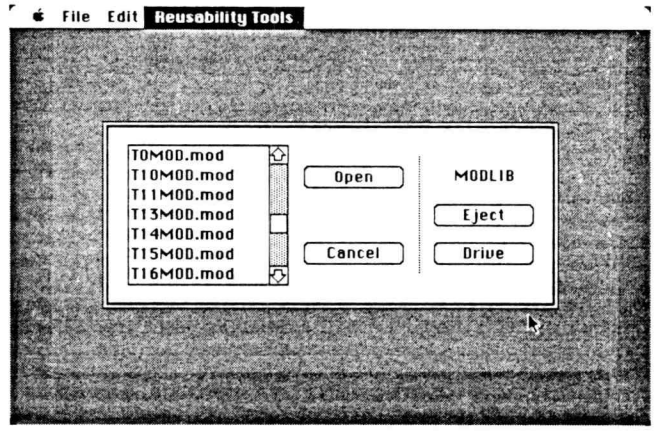


Figure 2.4: MCUIR Module Selection Dialog

Figure 2.7 is an example of a dialog box and the rule generated by the User Interface Routines of another instance of Yashar, Generalizer/Refiner. In this instance of Yashar, the format of restructuring of Modula-2 constructs are decided at will during the execution time. In Figure 2.7, the user selects to replace the conditional part of Modula-2 IF statements by meta identifiers; for the Generalizer/Refiner instance of Yashar, a meta identifier is a string of cardinal numbers prefixed by ##. The following is the rule generated and loaded to the rule repository automatically by the User Interface Routines of Generalizer/Refiner for the selection in Figure 2.7.

16:IF @01%06 THEN ...END

Afterwards, anytime the rule processor encounters a Modula-2 IF statment, it will activate a user-defined function from the *Function Table*, in this case function number 6, to create the appropriate meta identifier and to replace the conditional part of the IF statement with the created meta identifier. An IF statement after such a transformation would look like:

IF ##1 THEN ...END

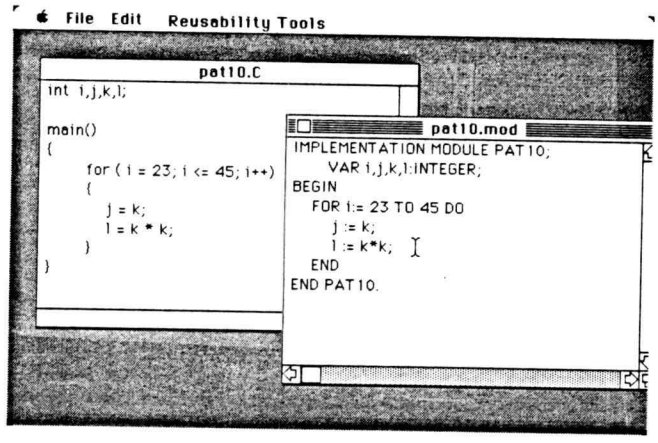


Figure 2.5: A Simple Modula-2 program and its C equivalent

2.5 Document Parser

Yashar uses the notion of a *hierarchical software document* as its input data model [KS83]. A hierarchical software document is a tree representing the objects that are to be processed by a tool. For example, source programs can be easily converted to a tree representation before being manipulated by Yashar's rule processor. Figure 2.8 shows a Modula-2 source code program after it is converted into a tree structure by the Document Parser.

A special purpose Document Parser must be hand-crafted for every language processed by Yashar. To facilitate this work, there is a set of built-in Yashar support routines that provide primitive operations for creating and manipulating a tree, see Appendix B. This reduces the designer's task to deciding the most logical order of creating tree nodes and saving the attributes of each node in the data dictionary. For a complete definition of the tree representation for Modula-2 refer to Appendix C.

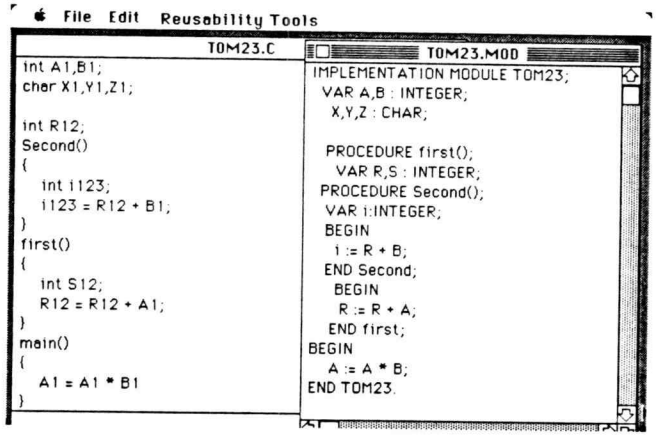


Figure 2.6: A Nested Modula-2 program and its C equivalent

2.6 Yashar's Rule Processor

All transformational operations are carried out by interpreting either designer-defined rules or run-time defined rules. The run-time defined rules are installed in the Rule Repository *on the fly* by User Interface Routines. A run-time defined support function may modify an existing designer-defined rule, or write an entirely new rule.

The rule processor consists of 1) a Function Table, 2) a Rule Repository, and 3) a Scratch Pad Area. The *Function Table* holds the address of designer-defined functions. The purpose of the designer-defined functions is to extend the functionality of rules. Designer-defined functions must be written to support special cases that cannot be accomplished by rules alone. For example, in the Modula-2 to C translator a designer-defined routine is needed to resolve the scope problem stemming from non-local variables referenced in nested procedures that are de-nested in C. Figure 2.6 is a case in which for resolving the scope of identifiers when de-nesting the procedures a designer-defined routine was activated by the rule processor in the course of performing the instructions of the rules for translation of Modula-2

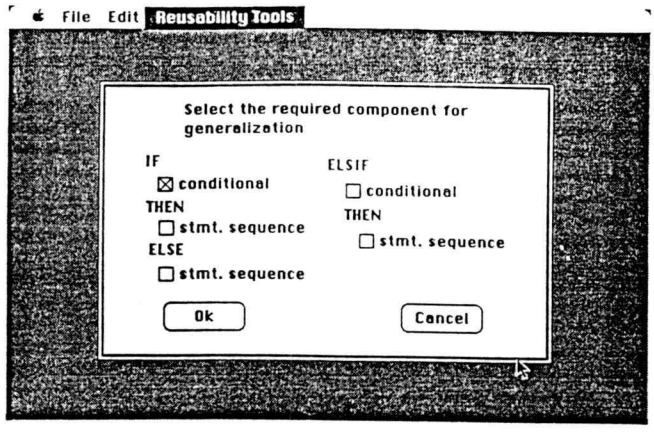


Figure 2.7: Interface to generate rules automatically

programs.

The *Rule Repository* is the storage element where designer-defined and run-time defined rules are stored and accessed by the rule processor.

When processing the tree-structured input, designer-defined rules define the default sequence of activities of the rule processor. For example, in an adaptive prettyprinter, designer-defined rules govern the traversal of the tree and tell how to produce the formatted output text.

Run-time defined rules modify or augment existing designer-defined rules. A rule can be modified more than once and the most recent modification is the one which is used by the rule processor. Furthermore, a modification to a designer-defined rule can be reversed. This feature provides the capability of processing the tree representation of input data in a variety of ways depending upon the programmer's expectations. For example, a tool created using Yashar for program reading allows the programmer to hide or unhide portions of the program source at will. The hiding and unhiding is easily achieved through modification of some of the rules of this tool responsible for producing the textual representation of the subtrees, (see

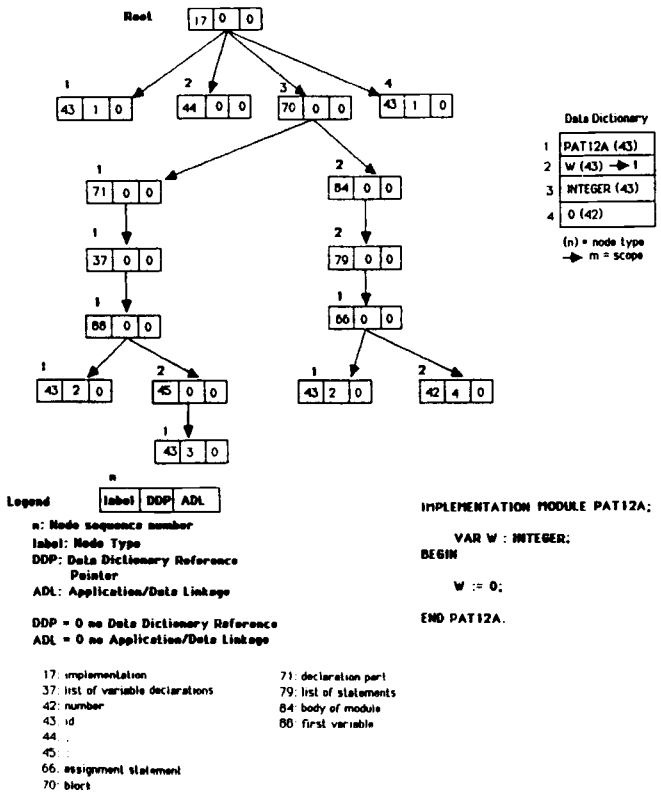


Figure 2.8: Internal representation of an object in Yashar

example, page 29).

The *Scratch Pad Area* is a set of *Registers* used to communicate among designer-defined functions, between designer-defined functions and the rule processor, and among the rules themselves. These registers can be accessed either from within rules or from designer-defined functions by calling Yashar built-in routines. In restructuring applications there are often cases in which traversing a subtree either should be delayed or repeated more than once. This could easily be done by storing the address of such a subtree in a register and using that register later to retrieve the desired information. For example, in translating FOR-loops from Modula-2 to C there is a need to have access to the indexing identifier of the FOR-loop in three different circumstances: 1) for initializing the indexing identifier; 2) for generating the terminating condition; and 3) for updating the indexing identifier(i.e.: incrementing or decrementing it). The indexing identifier is accessed by storing

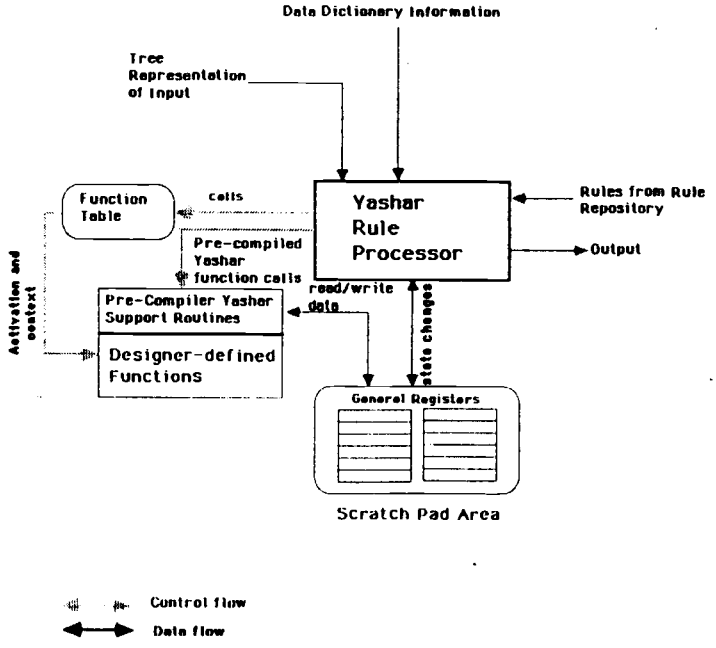


Figure 2.9: Yashar's Rule Processor Execution Environment

the address of the subtree in one of the registers and referencing it in appropriate locations. Consider the following FOR-loop in Modula-2, and its unparsing rule 14. This rule is a designer-defined rule as part of prettyprinting rules written for unparsing tree representation of Modula-2 source programs.

$$FOR\ i\ :=\ 1\ TO\ 10\ DO \ldots END$$

$$14:\ FOR\ @01\ :=\ @02\ TO\ @03\ DO \ldots END$$

@01, @02 ,and @03 represent the indexing identifier (i), initializing expression (1), and terminating expression (10) subtrees respectively. The rule for translating the above Modula-2 FOR-loop to C, and its translation would be as follows:

$$14: for(\ @M\$R03 = (@01)\$\ @01 = @02;\ @XR03\ <=\ @03; @XR03++)\{\ldots\}$$

$$for(i = 1; i <= 10; i++)\{\ldots\}$$

The above rule for translating Modula-2 FOR-loop to its C equivalent is also a hand written, therefore designer-defined rule. In the rule definition given above, the address of the subtree referring to the indexing variable is saved in register number 3 (@M$R03=(@01)$), and later is traversed, creating the terminating condition (@XR03 <= @03), and updating the index variable (XR03++).

As another example of a designer-defined (hand written) rule for translating Modula-2 FOR-loop into its semantically equivalent C construct using *while* statement, and the translation of the above FOR-loop example would be as follows:

$$14: \ @M\$R03 = (@01)\$ \ @01 = @02; @nwhile(@XR03 <= @03)\{\ldots @XR03++\}$$

$$i = 1;$$

$$while(i <= 10)\{\ldots i++\}$$

Notice the node label for Modula-2 FOR-loop subtrees is 14. The rules for restructuring the FOR-loop subtrees must also use the same label number. For processing tree nodes, the rule processor always searches the rule repository for a rule definition with the same label number as the tree node. If there does not exist a rule definition for the node under process, the rule processor discontinues processing the tree and the cause is communicated with the user through the User Interface Routines of the instance of Yashar. For a more detailed explanation of the rule instructions see Appendix A.

Tree Traversal

The rule processor traverses the tree and processes each node in the tree according to navigational and operational directives that are specified in each rule. When a node is visited, the repository is searched for a rule with a corresponding label. Then the rule is applied to the tree node and the next node to be visited is determined by the rule. The minimum sequencing instruction for each rule is @* which causes the tree to be traversed in *depth first* order.

For example, the following rule directs the rule processor to ignore the second child of the tree node with label 39 and recursively process its first, third and fourth children respectively.

39 : @01@I02@03

The rule processor reads this rule and does the following:

- Process the first child of node labeled 39 (@01)
- Do not process the second child of node labeled 39 (@I02)
- Process the third child of node labeled 39 (@03)

The above sequence of activities applies to all nodes with label 39. For a complete list of the instruction set of Yashar's rule processor and their meaning refer to Appendix A.

Rule Syntax

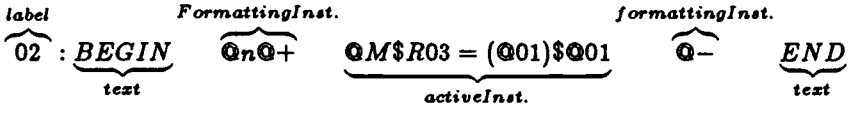
Each rule is a mixture of text, active and passive instructions. To distinguish between instructions and the text that is passed along, instructions are prefixed by an @ symbol. The components of a rule are:

- a label, always
- one or more active instructions, always
- text, optionally
- one or more formatting instructions, optionally

The label designates the type of node to be operated on by the rule processor. The rule is applied to *all* nodes of the type specified by the label.

Active instructions are responsible for: 1) sequencing the processing order of tree nodes, and 2) providing a mechanism for communicating data and control values among the rules, built-in Yashar functions, and designer-defined functions.

Formatting instructions and text do not have any effect on tree nodes, and serve only to format the output. For example the following rule:



directs the rule processor to transform every tree node with label 02 as follows:

- Emit a **BEGIN** (BEGIN)
- Emit a newline symbol (@n)
- Increment the indentation level by one increment unit (@+). An *increment unit* is assumed to be four character positions, the default value can only be altered prior to activation of the rule processor.
- Save the address of the first child of the current node in register R03 (@M$R03=(@01)$)
- Process the first child of the current node (@01). To make the rule processor operate on a specific child of a node, a child's sequence number is used. Thus @01 designates the first child of every tree node labeled 02.
- Decrement the indentation level by one increment unit (@-)
- Emit an **END**

The terse notation of the rules is not very human readable, instead, they are designed for machine processing, much like assembly language. Eventually there will be a higher level notation, and special interfaces for entering new rules.

It is important to note the difference between rules in Yashar and print specifications that are used in syntax directed editors such as PECAN [Rei85]. Like definitions in a syntax directed editor, Yashar rules can specify syntactic structure, but in addition Yashar rules specify semantic and deep structure of the information stored in the tree. For example, consider the case in a language directed editor where it is desired to hide the details of certain sections of code to avoid cluttering the focus of attention. The following rule defines the processing of a *while loop* to show only the predicate and number of statements of its body rather than showing all the statments of its body.

$$38 : WHILE@C+@01@D/;@n/@C-DO@+@n\overbrace{@An/<whilebody>/@02}^{\textit{added for hiding details}}@n@-END$$

This rule suppresses the details of the following while-loop:

Before	After
Detailed While loop	While loop abstraction without details of its body

WHILE a <= b DO
a := a +1;
IF b <> 0
THEN ...
⋮
END
	WHILE a <= b DO
	<whilebody> 10
	END

The following is an explanation of the rule:

- Emit a **WHILE** (WHILE)
- Activate conditional filling (@C+)
- Process the first child of the current node (@01)
- Emit a semicolon (;) and newline (@n) after processing of each child of second child of current node (@D/;@n/). Note that in (@D/;@n/) the slashes (/) are used to enclose the delimiter pattern.

- Turn off conditional filling (@C-)
- Emit a **DO** (DO)
- Increment the indentation level by one increment unit (@+)
- Emit a newline symbol (@n)
- Abstract the second child of current node and return <whilebody> and number of statements (children) of second child of current node (@An/<whilebody>/@02)
- Emit a newline symbol (@n)
- Decrement the indentation level by one increment unit (@-)
- Emit an **END** (END)

Formatting Instructions

Formatting instructions are for prettyprinting the textual representation of the input tree. These instructions do not effect the state of the nodes of a tree. For example @n, @+, and @- cause the rule processor to emit the control sequence to generate a new line, increment indentation level, and decrement the indentation level respectively.

$$02 : BEGIN \quad \overbrace{@n@+}^{Formating\ Inst.} \quad @01 \quad \overbrace{@-}^{Formating\ Inst.} \quad END$$

Therefore the above rule causes the rule processor to do the following:

- Emit **BEGIN** (BEGIN)
- Emit a newline symbol (@n)
- Increment the indentation level by one increment unit (@+)

- Process the child number one (@01)
- Decrement the indentation level by one increment unit (@-)
- Emit an **END** (END)

Escape and Break Point Instruction

The % symbol designates an *escape* instruction. If a navigation instruction has an % appended to it, the rule processor will execute the function referenced by the next two digits instead of processing the node referenced by the navigation instructions. The two digit number following % is an index into the Function Table which selects the designer-defined function to be executed. For example the following directs the rule processor to skip the second child of every node whose label is equal to 34 and to pass control and context of the rule processor to the activation of 5th function in the Function Table.

34 : @01@02%05

The **@Jn** designates a designer-defined function call instruction. The rule processor executes the *nth* function in the function table. In contrast to (**%n**), when using **@Jn** the context information is not passed to the called function.

Yashar's rule processor supports the insertion (definition, @P) , activation (@Z), and removal of break points (@V) to temporarily interrupt the processing of a rule. One can use the break point facility to step through a class of tree nodes (for a more detailed explanation of these instructions see A.2).

Arithmetic and Relational Instructions

Arithmetic operations use Scratch Pad Registers and constant values. The binary arithmetic operators +, -, *, /, and relational operators ==, >=, <, <=, != are supported.

$$@M\$R03 = (R02 + R07)\$$$

This rule assigns the sum of the values stored in register two and seven to register three. The precedence and order of evaluation is the same as for the C language [KR78].

Miscellaneous Instructions

The instructions to manipulate the rule repositories, access the data dictionary information, etc. belong to this category. For example:

$$30 : @m16\$IF@01\%12THEN@ + @D/; @n/@ * /@n/@n@ - END\$ \ldots$$

will cause the rule processor to first modify the rule definition of the tree nodes labeled 16 to what is enclosed between two $ delimiters, and then continue with the remainder of the rule definition for tree nodes with label 30. To restore the original definition of rule for nodes labeled 16, some rule must contain ...@r16. Alternatively, a built-in Yashar support routine can be called from within designer-defined functions to change the definition of the rule for nodes labeled 16 back to its original form.

Rules vs. Statements

It is important to differentiate the notion of rules from programming statements. In a programming language the sequence of statements are important, whereas in Yashar there is no sequencing implied by the ordering of the set of rules. Each rule defines a transformation independent of all other rules.

There is, however, a processing sequence established by the rules in much the same way as a logic program written in Prolog establishes a sequence. This sequence

depends on the shape of the tree and the operations given in the rules. In this sense, Yashar rules are analogous to Horn clauses in logic programming.

2.7 Creation of An Instance of Yashar

An instance of Yashar is created by a designer who must tailor Yashar to a specific application. Since Yashar is written in C, and the designer must provide a small number of C support routines, the steps in Figure 2.10 involve the C complier and linker. This is a *one time only* process which we call *instantiation of a tool.*

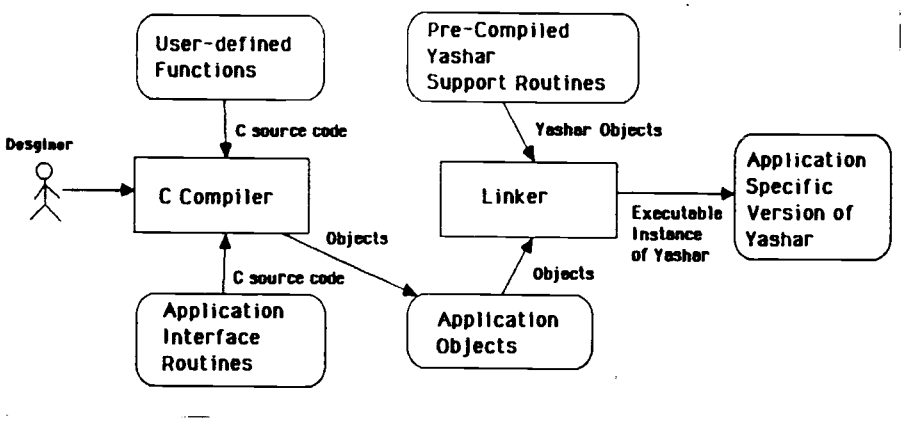


Figure 2.10: Building of a Tool

First, the tool designer must write User Interface Routines. This usually involves writing C functions to handle windows, dialogs, and menus. For example, the Modula-2 to C Translator User Interface takes care of selecting the Modula-2 source code to be translated, activation of the Modula-2 Parser, and requests to translate.

Next, the designer must write designer-defined rules which are used by the Rule Processor. For example, the rules in Modula-2 to C translator define the sequence of operations that re-write a Modula-2 program into a C equivalent.

Finally, designer-defined routines must be written for each special cases that can not be handled by the instructions of the rule processor. In the Modula-2 to C translator, a designer-defined routine is needed to resolve the scope problem stemming from non-local variables referenced in nested procedures that are de-nested in C.

Finally, the Document Parser must be written if one does not already exist for a given language. This could be done through automatic parser generators such as YACC [Joh75] or similar tools.

2.8 Conclusions

Yashar began as an experiment in building tools for program transformation [BGW76,Che83] and to study reusability of existing programs in block structured languages. The initial approach was based on the idea of unparsing in structured program editors [TR80,DVHK80]. The syntax of Yashar's rules were adapted from [Fri83] which in turn were borrowed from ALOE of GANDALF [NH81]. We have extended [Fri83] and the concept of a rule-based meta-tool so that a broader range of tools can be quickly created. The instruction set of Yashar's rule processor is powerful enough to manipulate structured data, making Yashar useful in program development environments. The terse notation of Yashar's instruction set facilitates automatic generation of rules by interface programs. The capability of modifying rules during execution is valuable in adjusting the actions of tools generated by Yashar therefore adding to their versatility. Furthermore, escaping the ordinary sequence of processing of tree nodes through escape and break point instructions makes the rule processor more functional. The relative simplicity of modifying the rules, with almost no overhead, makes tool building with Yashar an attractive alternative to traditional methods of 100% coding. Modification and maintenance of tools built using Yashar usually require changing the definition of some of the rules. In ad-

dition, Yashar offers a high degree of flexibility in building transformational and re-structuring tools. For example, one major advantage of using Yashar in building source language to source language translators is the ease of accommodation of dialectical variances for both source and target languages.

One of the drawbacks of Yashar is that the current syntax and notation of rules are not very readable for humans. As mentioned earlier the reasons for that were: 1) to allow automatic generation of rules; 2) ease of interpretation; and 3) to avoid parsing effort. Manual creation of the User Interface Routines and the writing of designer-defined routines may seem to be a drawback. However, this is nominal and well worth the effort considering that an entire family of tools are obtained as a result.

2.9 References

[BGW76] R. Balzer, N. Goldman, and D. Wile. On The Transformational Implementation Approach To Programming. In *Proceedings of the Second International Conference on Software Engineering*, IEEE Computer Society, Long Beach, Calif., 1976.

[BL86a] Abbas Birjandi and T. G. Lewis. *Arash:A Re-Structuring Tool For Building Software Systems From Reusable Components.* Technical Report 86-10-2, Department of Computer Science Oregon State University, Corvallis Oregon 97331, 1986.

[BL86b] Abbas Birjandi and T. G. Lewis. *Artimis: A Module Indexing And Source Program Reading And Understanding Environment.* Technical Report 86-10-3, Department of Computer Science Oregon State University, Corvallis Oregon 97331, 1986.

[Che83] T.E. Cheatham. Reusability Through Program Transformations. In *Proceedings of Workshop on Reusability in Programming*, pages 122–128, The Media Works, Inc., Newport, RI, September 1983.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog. Texts and Monographs in Computer Science*, Springer-Verlag, New York, 2 edition, 1984.

[Col85] Alain Colmerauer. Prolog in 10 figures. *Communication Of The ACM*, 28:1296–1324, December 1985.

[DKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Melese. Document Structure And Modularity In Mentor. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical*

Software Development Environments, pages 141–148, ACM, Pittsburgh,Pennsylvania, April 1984.

[DVHK80] V. Donzeau-Gouge, Veronique, Huet, and G. Kahn. Programming Environment Based On Structured Editors:The Mentor Experience. In *Workshop on Programming Environments*, Ridgefield, CT, June 1980.

[Fel79] S. I. Feldman. Make–a Program For Maintaining Computer Programs. *Software Practice And Experience*, 9(4):255–266, 1979.

[Fri83] Peter Fritzson. *Adaptive Prettyprinting of Abstract Syntax Applied to ADA and PASCAL.* Technical Report, Department of Computer Science, Linkoping University, Linkoping, Sweden, September 1983.

[Joh75] Stephen C. Johnson. *Yet Another Compiler-Compiler.* Technical Report, Bell Laboratories, Murray Hill, N.J., 1975.

[KM81] Brian W. Keringhan and John R. Mashey. The Unix Programming Environment. *IEEE Computer Magazine*, 14(4):12–24, 1981.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language. Prentice-Hall Software Series*, Prentice-Hall, 1978.

[KS83] Gary D. Kimura and Alan C. Shaw. *The Structure of Abstract Document Objects.* Technical Report, Computer Science Dept. University of Washington, Seattle, Washington, September 1983.

[NH81] David S. Notkin and Nico Habermann. Software Development Environment Issues As Related To Ada. In *Tutorial:Software Development Environments*, pages 107–137, IEEE Computer Society, 1981.

[Ost81] Leon Osterweil. Software Environment Research: Directions For The Next Five Years. *IEEE Computer Magazine*, 14(4):35–43, 1981.

[Rei85] Steven P. Reiss. Program Development Systems That Supports Multiple Views. *IEEE Transaction on Software Engineering*, SE-11(3), March 1985.

[TM81] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. *IEEE Computer Magazine*, 14(4):25–33, 1981.

[TR80] Tim Teitelbaum and Thomas Reps. *The Cornell Program Synthesizer:A Syntax-Directed Programming Environment.* Technical Report, Computer Science Dept. Cornell University, MAY 1980.

Chapter 3

Arash: A Re-Structuring Tool For Building Software Systems From Reusable Components

Arash: A Re-Structuring Tool For Building Software Systems From Reusable Components

Abbas Birjandi

T.G. Lewis

Department of Computer Science

Oregon State University

Corvallis, Oregon 97331

(503) 754-3273

Abstract:

Arash is a rule-based tool which can be applied to a family of programming languages, e.g. Pascal, Modula-2, and C. In this paper we describe how to use Arash to restructure Modula-2 source code modules taken from a programmer's database of reusable components in order to construct new software systems, quickly and correctly. Arash incorporates a collection of Generalizers which transform source code modules into abstracted modules. Conversely, a collection of Refiners produce a concrete instance from an abstracted source module. Both Generalizers and Refiners operate on source code components called *fragments* to restructure existing programs, documentation, and associated text.

Keywords: Programming environment, program transformation, source code mutation, syntax directed tools, code fragments, code selection, customizing, general software, generic systems, program generation, tailoring, reuse of software.

3.1 Introduction

Reusability is defined as anyway in which previously written software can be used for a new purpose or to avoid writing new software [Ker83]. This definition covers reuse of software at both object code and source code level. The potential benefits of reusing existing software are: 1) reduction in the cost and development time to produce a new program or system of programs, and 2) an increase in the ease of maintenance and enhancement of existing software systems [Che83].

Arash is a tool that operates directly on source code. Reuse of source code in contrast to object code has the advantage of 1) adapting the interface as well as implementation part of a module to a new interface specification, 2) providing an opportunity to tune, optimize, and eliminate unnecessary code, and 3) providing readable code so that a programmer's knowledge of the reusable module is increased. This allows the possibility of:

1. Source code reuse/replication by reuse of part or all of existing source code or its data structure,

2. Detailed algorithm reuse by reuse of source code from existing programs as an example of how to do a new program,

3. Large-scale structural reuse by selecting and adapting program structure,

4. Maintainability/enhanceability by increasing the effectiveness of programmers by enabling them to study programs with the aid of understandability tools,

5. Portability by facilitating the reuse of software across a wide range of hosts, and

6. Optimization by enabling tuning of generated source code.

The basic idea in Arash is to construct new program modules from old ones by applying two automatic transformations: *Generalization* and *Refinement*. Generalizers transform a source code module to an abstracted form. Refiners operate on the abstracted form of a module to produce a concrete instance. Figure 3.1 shows a Modula-2 source program (**sort1.mod**) for sorting an array of integers, its abstracted form produced automatically by a series of Generalizers (**sort1.GLS**), and a concrete instance to sort an array of strings (**sort1.NEW**) generated by a series of Refiners. Notice in the abstracted version the actual abstracted program fragments are replaced by special identifiers called *meta identifiers* (e.g. ##1, ##2,...).

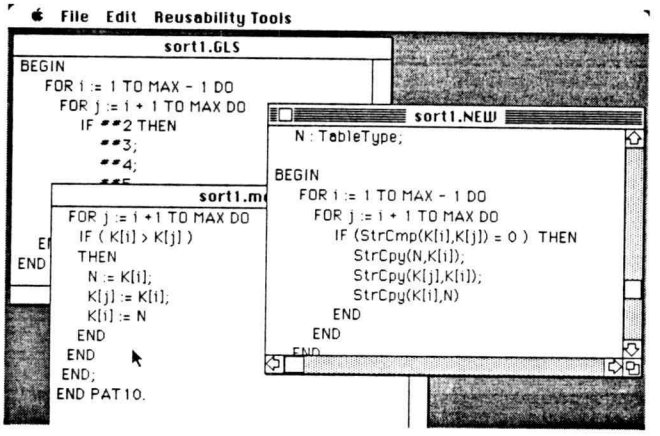


Figure 3.1: Sort Before and After Generalization and Refinement

Although other source languages might be used with Arash, Modula-2 [Wir83] is used as the source language. In Modula-2 , a module has two parts: 1) a definition part which defines the constants, types, variables, and procedures of the module which can be accessed by other modules and, 2) an implementation part that encapsulates the actual implementation detail of the module. In addition, Arash supports the concept of an extended module—one that has attributes. The additional attributes needed for reusability are *.GLS* which contains a reusable module with meta identifiers replacing source code fragments, *.DRI* which contains a list of

meta identifiers and the actual code fragments they replaced, *.MLS* which contains the rules that are used for the process of generalization, and *.NEW* which contains the newly generated concrete module.

Two other components of this tool are a *Programmer's Data Base* (GrabBag) used to search for reusable source programs, and a set of *Browsers* to aid in reading and understanding the existing source programs. The Programmer's Data Base and Browser facilities are the subject of another paper and will not be discussed in this paper [BL86b].

Related Work

A related project in reusability through program transformation is PDS [Che83]. PDS is an integrated programming support environment that has three major components: a software database, a user interface, and a collection of tools that can be called via the user interface to manipulate the software modules stored in the software database. PDS uses an extended version of EL1 programming language. EL1 is an extensible language and supports programmer–defined data types, generic routines, and programmer control over type conversion [Weg74]. The abstracted programs are built in EL1 and the transformation is done by defining transformational rules containing a syntactic pattern part , optionally augmented by a semantic predicate and a replacement. Both Arash and PDS use transformation based approach for creation of reusable components. Arash differs from PDS in that: 1) Arash operates on a family of languages; 2) Arash operates on structured data, tree representation of input; 3) Arash operates on existing software for creation of abstract components and their latter refinement to concrete versions. transformation in Arash are written in a special purpose language based on Arash uses a multiple window, menu based environment with graphical support to interact with user. Effort is underway to create similar user environment for PDS [Che83].

Reusability Life Cycle and Arash

When reusable components are used to build a new software system, the traditional software life cycle is altered. Table 3.1 shows the difference between traditional software life cycle and reusability life cycle. The additional phases in the reusability life cycle indicate how a designer uses existing components rather than implement everything from the beginning. Problem definition is the phase during which the

Traditional Life Cycle	Reusable Life Cycle	Arash Support
Problem Definition	Problem Definition	None
Requirement Analysis	Requirement Analysis	None
System Design Specification	Find and reuse similar System Design Specification	None
Detailed Design Specification	Find and reuse similar Detailed Design	None
Implementation	Find and reuse existing routines from object code library	None
	Find and reuse (modified) source code from previous systems	Generalizers,Refiners Grabbag,Browser
	Produce Glue Code	None
Testing	Testing	None
System Integration	System Integration	None
Maintenance	Reuse of original product	Generalizers, Refiners

Table 3.1: Reusability Life Cycle Stages vs. Traditional Life Cycle

problem to be solved is formalized as a set of needs; requirement analysis is the process of studying user needs to arrive at a definition of system software requirements; system design specification is the period of time during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirement; detailed design specification is the period of time during which the design of system or a system component is documented; typical

contents include system or component algorithms, control logic, data structures, data set-use information, input/output formats, and interface description; implementation is the period of time during which a software product is created from design documentation and debugged; testing is the period of time during which the components of a software product are evaluated and integrated to determine whether or not requirements have been satisfied; system integration is the period of time during which a software product is integrated into its operational environment and tested in this environment to ensure that it performs as required; maintenance is the period of time during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements. A component is a basic part of a system or program; an interface is a shared boundary to interact or communicate with another system component [Sta83a]. Glue code is the minimal extra code that may be needed to bring the reused modules together. Arash is only applicable for reusing and maintenance of existing source programs. Maintenance may be considered as reusing the original product [Fre83]. In maintenance, problem specification is usually better defined and the reusable module does not have to be found [Fre83].

3.2 Arash System Architecture

The overall structure of Arash is shown in Figure 3.2. A programmer interacts with Arash through requests which are processed by the Arash User Interface Routines (AUIR). These routines provide a user interface (windows, icons, menus, and a mouse), exercise control over the activation of Generalizers and Refiners, and communicate with the Rule Processor. Figure 3.3 shows what the user interface looks like when running Arash.

A Modula-2 source program is read from a text file by the Modula-2 Parser,

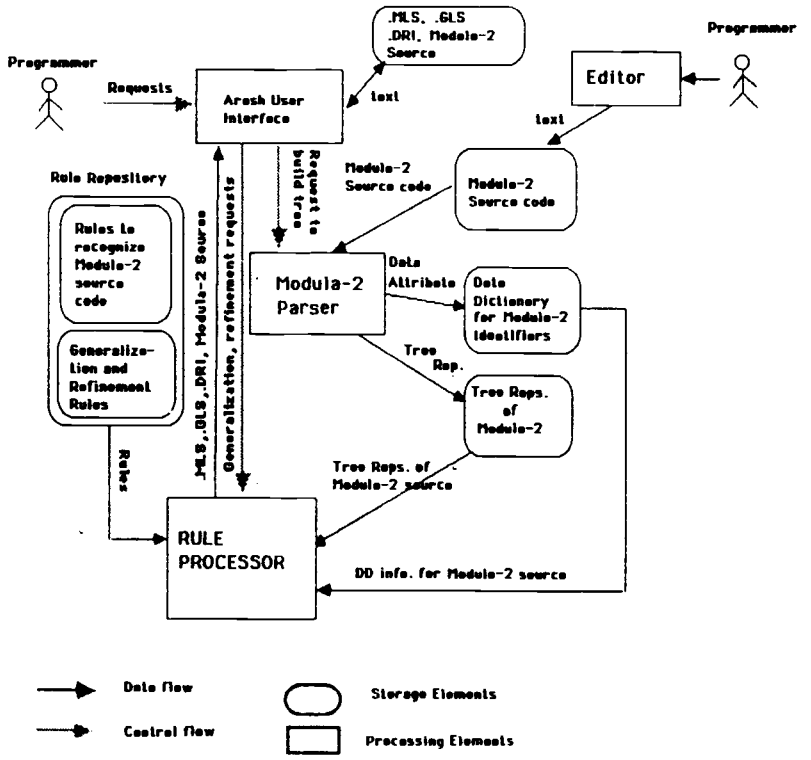


Figure 3.2: Arash System Structure

shown in Figure 3.2. Symbol table information is stored in the Data Dictionary, and the transformed source program file is stored in the tree representation of Modula-2 data structure. This internal data structure is processed by the Rule Processor.

When a user requests that the tree representation of a Modula-2 program be generalized, a collection of AUIR's are activated which traverse the tree and produce .MLS, .DRI, and .GLS files. The Rule Processor takes a rule from the Rule Repository, processes it, calls the appropriate AUIR, and outputs the result to the .MLS, .DRI, and .GLS attribute files.

Similarly, when the user requests that the .GLS file of some program be refined, the Refiner uses rules from the Rule Repository to carry out a refinement. The Rule Processor returns the result of generalization and refinement (.GLS, .MLS, .DRI,

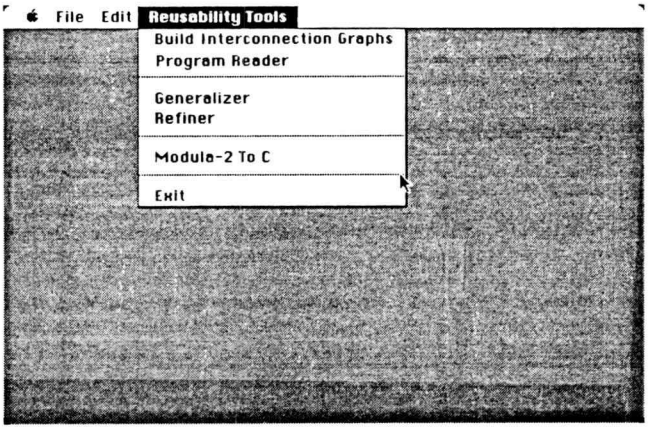


Figure 3.3: Arash User Interface Options

and .NEW) to AUIR which in turn displays each file in a text editing window so it may be inspected and saved by the user. These files are described below:

Attribute File

The .DRI attribute file contains the rules used by the rule processor to transform a source program into an abstraction. This file is created by Generalizers and used for refining a module to a concrete form. The rules in a .DRI file are copied into the rule repository by AUIR prior to refining an abstracted module to a concrete one.

The .MLS attribute file is generated by the rule processor during the generalization of a source program. It contains a list of meta identifiers and the actual source code that each meta identifier replaced. For example, see Figure 3.4. The .MLS file is used by the Refiner to replace meta identifiers with actual source code. The contents of a .MLS file may be modified either by a rule, or by manual editing of the file prior to Refinement.

For each source program module which is abstracted by the Generalizers a .GLS attribute file is created which contains a textual representation of the abstracted

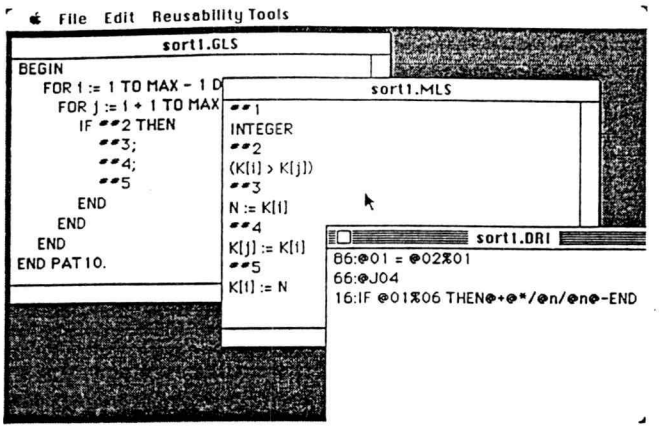


Figure 3.4: Attribute Files Generated by Generalizers

module. It serves as visual feedback to the user to verify the operation of the Generalizers and has no other significance.

Internal Structure of Arash

To understand how Arash works, one must understand three central structures:

- The Internal Tree and Data Dictionary containing Modula-2 source program data. All inputs to Arash are first converted to a tree structure by the Modula-2 Parser as shown in Figure 3.2. Figure 3.5 shows a Modula-2 source code program as it is converted into a tree structure by the Modula-2 Parser.

- The Rule Repository , Rule Processor, and Syntactic and Semantic structure of Rules. The *Rule Repository* is the storage element where rules are stored and accessed by the rule processor. This storage element is divided into two logical parts. The first part contains rules to recognize Modula-2 source programs and the second part stores Generalization and Refinement Rules.

- The interaction among Rules, Rule Processor, and the Internal Tree/Data Dictionary. Rules to reconstruct Modula-2 source programs from the tree

representation of Modula-2 source programs are copied into the rule repository by the rule processor prior to processing the tree representation. See Appendix E for a list of these rules. Generalization of Modula-2 source programs and the refinement of the abstracted modules to concrete ones are done by a set of rules that are either generated by Generalizers, or supplied by the user through creation or modification of .DRI or .MLS files. Appendix D lists functions which produce generalization and refinement rules, automatically.

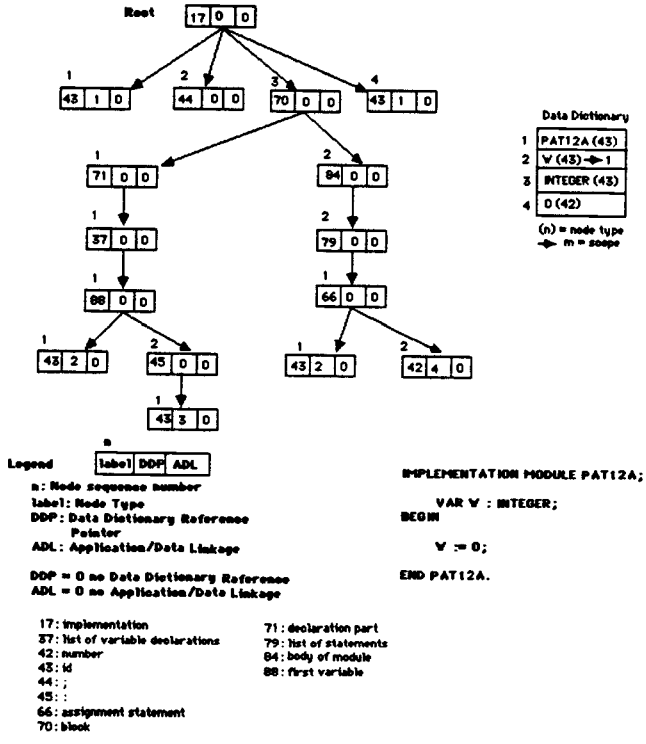


Figure 3.5: Internal representation of an object in Arash

The Rule Processor

The *Rule Processor* is a transformational unit which converts Modula-2 source code stored in the tree representation into either Generalized or Refined output,

See Figure 3.6. The rule processor consists of 1) a Function Table, 2) a Rule Repository, and 3) a Scratch Pad Area.

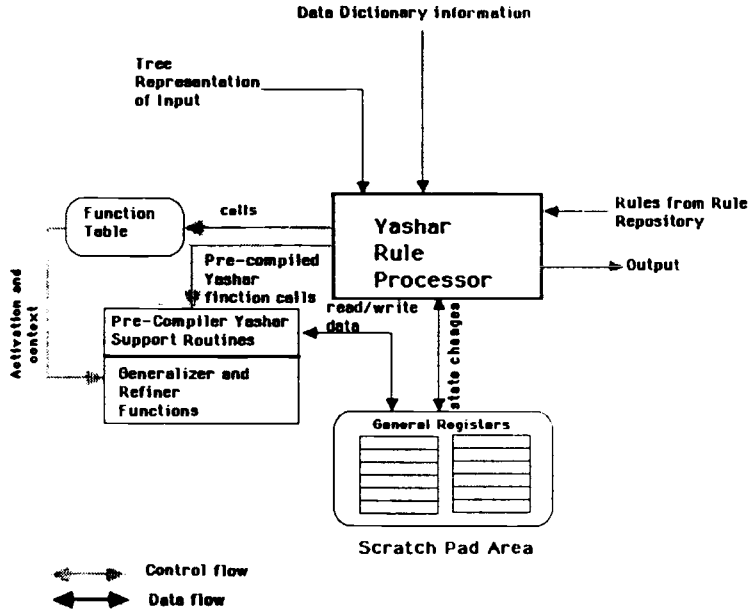


Figure 3.6: Arash's Rule Processor Execution Environment

The *Function Table* holds the address of designer-defined functions. Designer-defined functions are to extend the functionality of rules. Designer-defined functions perform tasks that are not possible or very difficult to support by the semantics of the instructions of the rule processor. In Arash the designer-defined functions perform the extra semantic checking needed for proper transformation of different constructs, for a list of these functions please refer to D and D.2.

The *Rule Repository* is the storage element where designer-defined and run-time defined rules are stored and accessed by the rule processor. For example the rules to produce the abstract modules are created at runtime by the Generalizers, which in turn used by the rule processor to produce the abstracted module from the tree representation. These rules are also captured in .DRI attribute file. Similarly the Refiners use the rules associated with each meta identifier in .MLS file for refining

the abstracted module.

The *Scratch Pad Area* is a set of *Registers* used by functions that carry out generalization and refinement to communicate among themselves and the rule processor. These registers can be accessed either from within rules or in designer-defined functions by calling Yashar built-in routines.

Rule Processing

The rule processor traverses the tree and processes each node in the tree according to navigational and operational directives specified in each rule. When a labeled node is visited, the repository is searched for a rule with a corresponding label. Then the rule is applied to all tree nodes with the specific label. Furthermore the next node to be visited is determined by the rule. The minimum sequencing instruction for each rule is @* which causes the tree to be traversed in *depth first* order.

The order of placement of rules in the Rule Repository does not have any significance on the order of their execution. In other words there is no sequencing involved in the formation of a set of rules to perform a series of transformation–this differentiates the notion of the rules in Arash from ordinary programming statements. Rules can be viewed as a formalism for defining knowledge independent of the method of computation. The rules in Arash are declarative in the sense that there is no sequencing implied in the order in which the rules appear. On the other hand, Arash rules differ from the declarative rules used in logic programming [CM84] where a rule states a proposition corresponding to a logical implication [Col85]. Arash rules include imperative commands which operate directly on inputs, much like the operations in Lisp which operate directly on input lists.

Rule Processor Instruction Set

The instruction set of the rule processor is divided into the following:

- Tree Navigation
- Formating
- Escape and Breaking
- Register Manipulation
- Miscellaneous

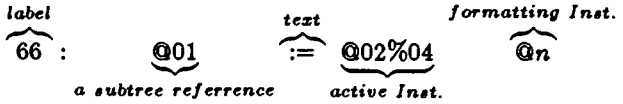
Rule Syntax

Each rule is a mixture of text, active and passive instructions. To distinguish between instructions and the text that is passed along, instructions are prefixed by an @ symbol.

The components of a rule are:

- a label, always
- one or more active instructions, always
- text, optionally
- one or more formatting instructions, optionally

The label designates the type of node to be operated on by the rule processor. The rule is applied to *all* nodes of the type specified by the label. Active instructions are responsible for, 1) sequencing the processing order of tree nodes, and 2) providing a mechanism for communicating data and control values among the rules and support functions. Formatting instructions and text do not have any effect on tree nodes, and serve only to format the output. For example the following rule:



directs the rule processor to perform the following on any subtree labeled 66 which is a subtree for assignment statement:

- Process first child, or at this case simply emit the identifier name (@01)
- Emit the character sequence := (:=)
- Emit the return result of activation of fourth function in the Function Table as the left hand of the assignment (@02%04)
- Emit a newline symbol (@n)

Notice the above scheme could be used to ensure type compatibility of assignment statements for numeric values. For example the refining function (04) in this case inspects the type information of the variable on the lefthand side of the assignment and creates necessary type casting structure for enforcing type compatibility. For detail explanation of the rule processor and the definition of the rules see [BL86c].

3.3 Generalizers and Refiners

Generalizers transform a Modula-2 source code component into a parameterized form called an *abstract module.* Refiners operate on the abstracted module to produce a concrete instance. Generalizer-Refiner pairs are inverse transformation operators.

A *program fragment* is a piece of source code representing the stereo-typical action sequence in programs [SE83]. Program fragments are meant to be modified or tuned to the particular task at hand [SE83]. For example a *WHILE* loop in a sort routine can be considered a loop fragment.

An abstract module is one in which certain *program fragments* are abstracted into a generic version by substituting a special identifier, called a *meta identifier* in the place of the program fragment. A meta identifier is a string of cardinal numbers

prefixed by ##. A concrete module is one in which meta identifiers are replaced by user defined or refiner generated text.

Generalizer Operation

A Generalizer transforms a set of program fragments $p_i \in P$ into a set of meta identifiers, $q_i \in Q$, where:

P Modula-2 source code module

p_i Modula-2 fragment

Q Abstracted module

q_i Modula-2 meta identifier

The transformation $G(P) \Rightarrow Q$ carried out by Arash Generalizers re-writes program P into meta-program Q through a series of generalizing functions:

$$G(P) = g_1 \cdot g_2 \cdot g_3 \cdots g_k(P)$$

The generalization functions $G(P) = g_1 \cdot g_2 \cdot g_3 \cdots g_k(P)$ are collections of rules which are automatically generated by the Arash User Interface Routines. The generated rules and the tree representation of source program P are passed to the rule processor where the instructions present in each rule perform the generalization.

G(P) produces three kinds of output: 1) .GLS attribute file with the abstracted Modula-2 reusable module containing *meta identifiers* in place of fragments, 2) .MLS attribute file containing a list of *meta identifiers* and the actual code that they replaced , and 3) .DRI attribute file containing the rules that were used to generalize P. Figure 3.4 is an example of these attribute files for the sort routine shown in Figure 3.1.

For example the rule in Figure 3.8 for generalizing the conditional part of *IF* fragments directs the rule processor to do the following anytime it encounters a tree node with label 16 while traversing the tree.

16:IF @01%06 THEN@+@D/;@n/@*@n@-END

- Emit an **IF** (IF)
- Instead of processing the conditional part (@01) call function number 6 in the Function Table (%06) and pass the *context* to it. Function number 6 in the Function Table is *MtIfG()* which is responsible for generating the meta identifier for the conditional part of any IF statement in the Modula-2 source program, and then saving the actual replaced code along with it's meta identifier in the .MLS file. See Appendix D for a list of functions which can be referenced through the Function Table.
- Emit a **THEN** (THEN)
- Increment the indentation level by one increment unit (@+)
- Set the delimiter string to ;@n (@D/;@n/)
- Process all the children of the node (@*)
- Decrement the indentation level by one increment unit (@-)
- Emit an **END** (END)

A user selects program fragments for generalization from a dialog, for example in Figure 3.7, a user has selected all *TYPE, ASSIGNMENT*, and *IF* fragments to be generalized. In Figure 3.8 the user further limits the generalization of *IF* primes to their *conditional* parts. Appendix F lists all menus used for generalization. Next, the user's selections are translated to a set of rules which define the functions $G(P) = g_1 \cdot g_2 \cdot g_3 \cdots g_k(P)$.

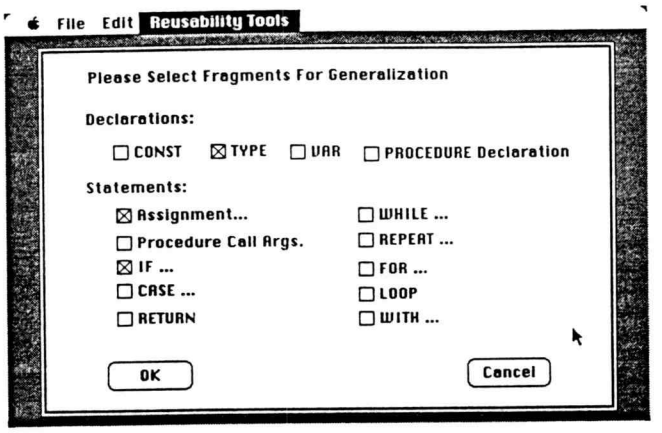


Figure 3.7: Dialog For Selection of Fragments

Refiner Operation

A refiner transforms a set of meta identifiers $q_i \in Q$ into a set of program fragments $p_i \in P$ where:

P Modula-2 source code module

p_i Modula-2 fragment

Q Abstracted module

q_i Modula-2 meta fragment

The transformation $R(Q) \Rightarrow P$ re-writes an abstracted module **Q** into a concrete program **P** through a series of Refining functions:

$$R(Q) = r_1 \cdot r_2 \cdot r_3 \cdots r_k(Q)$$

The $r_{i's}$ are refining rules that must be applied to a generalized module to create a concrete one. At the beginning of the refining session, AUIR 1) copies the rules in .DRI attribute into the Rule repository; these rules were generated by the Generalizers during the generalization; 2) the tree representation of the generalized

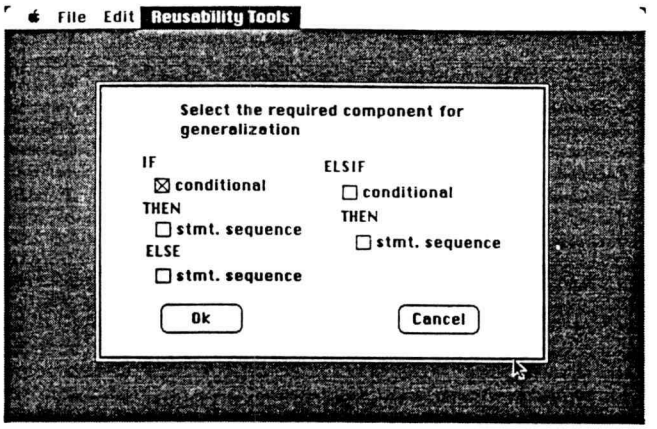


Figure 3.8: Dialog For Generalization of IF

module and its data dictionary information are prepared; and 3) the rule processor is activated. The rule which were loaded from .DRI will cause the rule processor to activate the appropriate refiner. Once a refiner is activated, it checks the .MLS file to find a match for the meta identifier representing an abstracted fragment in the tree representation of the generalized module. If there is a match the refinement is carried out, otherwise no action is taken and the actual code fragment is produced as if it has not been abstracted. The refined version of the module is captured in the window of .NEW attribute file for further inspection , editing or saving.

The meta identifiers in a .MLS file provide the Refiners with: 1) a check to see if a refinement should be done on the meta identifier that replaced the original fragment, and 2) a check to see if any rules should be modified in the Rule Repository before the refinement corresponding to a certain meta identifier. Manual deletion of any of the meta identifiers in the .MLS file has the effect of disabling the refinement process for that specific meta identifier. For example, the following will cause the original rule for nodes labeled 15 to be redefined prior to execution of the *Refiner* which operates on meta identifier **##1**.

$$\overbrace{\#\#1}^{\textit{meta identifier}} \quad : \quad \underbrace{15 : @J01@02\%10@J02}_{\textit{Redefinition of rule for nodes labeled 15}}$$

Consider the sort1.mod routine in Figure 3.1 which has been generalized and then refined into a routine to sort an array of character strings. The comparison and assignment operations are different for integers and strings, so the following rule in the .DRI file converts integer operations into string operations, see Figure 3.9.

86:@?(J02 == 1)??50:(StrCmp(@01,@02) = 0)@m66$StrCpy(@01,@02)@r66$?

Nodes labeled 86 are type declaration for array elements. Nodes labeled 50 are the conditional part of IF statements, and nodes labeled 66 represent assignment statements. When the rule processor encounters a node with label 86 it activates function number 2 (J02) which returns a 1 to specify an integer type and 0 to specify a character string type. If the evaluation of (J02 == 1) is true, meaning that on integer sort is desired, then no rule is modified, otherwise the rule definition for nodes with label 50 is modified to:

(StrCmp(@01,@02) = 0)@m66$StrCpy(@01,@02)@r66$

The modification of the the definition of the rule for tree nodes with label 50 causes the generation of the proper comparison construct for character strings and also modifies the definition of the assignment rule label 66 to generate the correct constructs for character strings. Figure 3.10 shows the sort routine generated by the refinement rule above. See [BL86c] for more details on the semantics of the rules.

3.4 Conclusion

Arash was built as an experimental tool to study reusability of software systems. Major goals of this effort were: 1) to use existing software components available

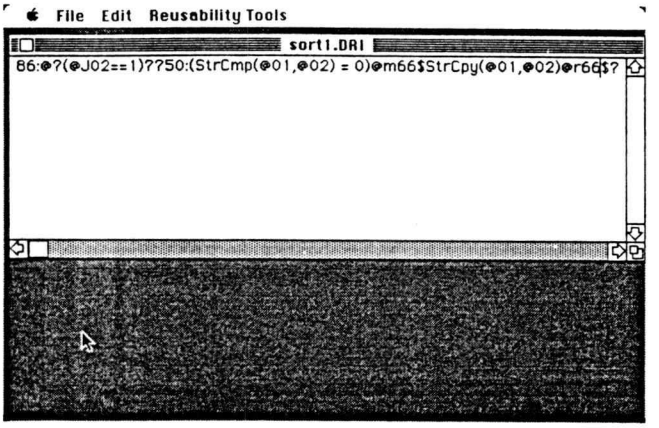


Figure 3.9: Rules For Re-Structuring Sort Fragment

in existing libraries of source codes, 2) to avoid creation of new programming languages and notations radically different from the majority of current software systems [Weg83,LM83] , namely existing block structured languages, which would discourage application of Arash, and 3) to follow the philosophy in which the creation of new software must occur automatically using notation which can be easily generated by computers.

Arash meets all of the goals: 1) it operates on a block structured language, 2) no new programming language is created, and 3) the rule based expressions for restructuring are easily generated and processed by computer.

Access to data dictionary information, flexibility of modifying rules interactively, and two escape mechanisms for semantic processing provide all the necessary tools for deriving a family of concrete programs from a single abstract program.

Often it is desirable to produce a tool without concern for its efficiency. This could be done from proper abstracted programs by defining and applying the correct refinements. This approach of deriving a tool quickly is often of value because it provides the opportunity to evaluate the specification of the tool by observing the behavior of its prototype.

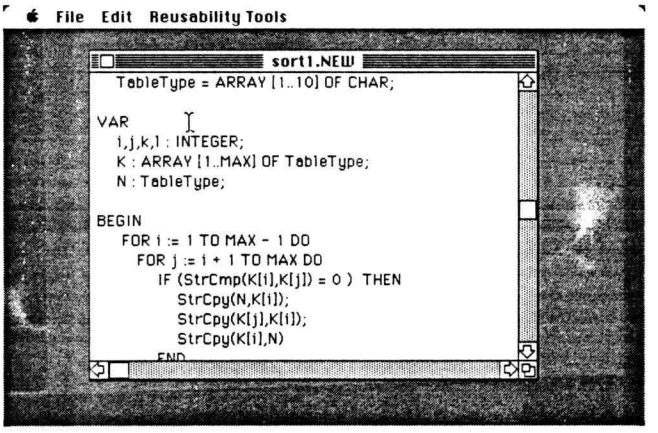


Figure 3.10: Re-Structured New Sort Fragment

Arash could also be used for tailoring an abstracted module for different targets, defining rules can be defined in a manner in which the differences among the targets are taken into account. Instrumentation and insertion of debugging codes is another application of Arash. Extra debugging codes can be generated by a refining process at will. The code can be eliminated later by reproducing the program *without* the rule definition that carried out generation of the extra debugging and instrumentation code. The fact that rules are capable of activating designer-defined routines provides the computational power of any typical programming language to the rule processor. This provides the basis for generating complicated transformers that would otherwise be impossible.

A limitation inherent to the class of languages Arash supports is the difficulty of mapping algorithms that use drastically different data structures, such as sorting a list of numbers stored in an array versus a linked list. This problem arises from the algorithmic differences between indexing through elements of an array and visiting elements of a linked list. Currently, Arash is not capable of restructuring an array dependent algorithm into an equivalent linked list dependent algorithm.

Another drawback is that current syntax and notation of rules are not very

readable for human. Reasons for this were to provide the ability of generating the rules automatically, to make their interpretation easier, and to avoid parsing effort which would otherwise be needed to reduce the overhead of experimenting with the definition of the rules and their modification.

Manual creation of the User Interface Routines and the writing of designer-defined routines may seem to be a drawback. However, this is nominal and well worth the effort considering that an entire family of tools are obtained as a result.

3.5 References

[BL86a] Abbas Birjandi and T. G. Lewis. *Artimis: A Module Indexing And Source Program Reading And Understanding Environment.* Technical Report 86-10-3, Department of Computer Science Oregon State University, Corvallis Oregon 97331, 1986.

[BL86b] Abbas Birjandi and T. G. Lewis. *Yashar:A Rule Based Meta-Tool For Program Development.* Technical Report 86-30-6, Department of Computer Science Oregon State University, Corvallis Oregon 97331, 1986.

[Che83] T.E. Cheatham. Reusability through program transformations. In *Proceedings of Workshop on Reusability in Programming*, pages 122–128, The Media Works, Inc., Newport, RI, September 1983.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog. Texts and Monographs in Computer Science*, Springer-Verlag, New York, 2 edition, 1984.

[Col85] Alain Colmerauer. Prolog in 10 figures. *Communication Of The ACM*, 28:1296–1324, December 1985.

[Fre83] Peter Freeman. Reusable software engineering:concepts and research directions. In *Proceedings on Workshop on Reusability in Programming*, pages 2–16, Newport, 9 1983.

[Ker83] Kernighan. The unix system and software reusability. In *Proceedings of the Workshop on Reusability in Programming*, pages 235–239, The Media Works, Inc., Newport, RI, September 1983.

[LM83] S.D. Litvintchouk and A.S. Matsumoto. Design of ada systems providing reusable components. In *Proceedings of Workshop on Reusability in Pro-*

gramming, pages 198–206, The Media Works, Inc., Newport, RI, September 1983.

[SE83] Elliot Soloway and Kate Ehrlich. What do programmers reuse? theory and experiment. In *Proceedings of Workshop on Reusability in Programming*, pages 184–191, The Media Works,Inc., Newport, RI, September 1983.

[Sta83] American National Standard. *IEEE Standard Glossary of Software Engineering Terminology*. New York, February 1983.

[Weg74] Ben Wegbreit. The treatment of data types in el1. *Communication of the ACM*, 17(5):251–264, May 1974.

[Weg83] Peter Wegner. Varieties of reusability. In *Proceedings of Workshop on Reusability in Programming*, pages 30–44, The Media Works, Inc., Newport,RI, 9 1983.

[Wir83] Niklaus Wirth. *Programming In Modula-2. Texts and Monographs in Computer Science*, Springer-Verlag, Berlin Heidelberg, 1983.

Chapter 4

Artimis:A Module Indexing and Source Program Reading And Understanding Environment

Artimis:A Module Indexing and Source Program Reading And Understanding Environment

Abbas Birjandi

T.G. Lewis

Department of Computer Science

Oregon State University

Corvallis, Oregon 97331

(503) 754-3273

Abstract:

Artimis is part of an environment for software reuse consisting of two logically independent portions, 1) the indexing and retrieval facility called, *GrabBag*, for storage and subsequent retrieval of reusable modules, and 2) a set of tools called *Browsers*, which aid reading and understanding of source programs. GrabBag creates a highly simple and friendly interface for retrieval of viable candidates for reuse. Browser's tool set, The Module Interconnection Graph Builder, Procedure Call Graph Builder, and Module Abstractor create different levels of abstraction to help a programmer understand a source program.

Keywords: Programming environment, program transformation, source code mutation, code fragments, code selection, program understanding, program reading, program maintenance.

4.1 Introduction

The reuse of existing software is seen as a measure of curtailing the high cost of software. The benefits of reusing existing software are: 1) reduction in the cost and development time to produce a new program or system of programs, and 2) an increase in the ease of maintenance and enhancement of existing software systems [Che83]. To reuse existing software one should know what existing software is available and how it can be used in relation to the task at hand.

Artimis is part of an environment for reusing software [Bir86] which provides a programmers database called *GrabBag* [San86] and a set of understandability and abstraction tools collectively referred to as *Browsers*. GrabBage provides a convenient way of locating a module and related documents called attributes. A module is an independent unit of code. Module attributes are known resources of a module such as a documentation file and an interface definition file.

Although other source languages might be used with Artimis, Modula-2 [Wir83] is used as the source language. In Modula-2 , a module has two parts: 1) a definition part which defines the visibility of constants, types, variables, and procedures of the module which can be accessed by other modules and, 2) an implementation part that encapsulates the actual implementation detail of the module.

Reusability

In [Ker83] reusability is defined as anyway in which previously written software can be used for a new purpose or to avoid writing new software. This definition covers representation of software at both object code and source code level. However, reuse of source code in contrast to object code has the advantage of 1) adapting the interface as well as implementation part of a module to a new interface specification, 2) providing an opportunity to tune, optimize, and eliminate unnecessary code, and 3) providing readable code so that a programmer's knowledge of the reusable module

is increased.

This allows the possibility of:

1. Source code reuse/replication by reuse of part or all of existing source code or its data structure,

2. Detailed algorithm reuse by reuse of source code from existing programs as an example of how to do a new program,

3. Large-scale structural reuse by selecting and adapting program design,

4. Maintainability/enhanceability by increasing the effectiveness of programmers by enabling them to study programs with the aid of understandability tools,

5. Portability by facilitating the reuse of software across a wide range of hosts, and

6. Optimization by enabling tuning of generated source code.

Reusability Life Cycle Vs. Traditional Life Cycle

When reusable components are used to build a new software system, the traditional software life cycle is altered. Table 4.1 shows the difference between traditional software life cycle and reusability life cycle. The additional phases in the reusability life cycle indicate how a designer uses existing components rather than implement everything from the beginning.

Maintenance may be considered as reusing the original product [Fre83]. In maintenance, problem specification is usually better defined and the product does not have to be located [Fre83]. Problem definition is the phase during which the problem to be solved is formalized as a set of needs; requirement analysis is the process of studying user needs to arrive at a definition of system software requirements;

Traditional Life Cycle	Reusable Life Cycle	Artimis Support
Problem Definition	Problem Definition	None
Requirement Analysis	Requirement Analysis	None
System Design Specification	Find and reuse similar System Design Specification	None
Detailed Design Specification	Find and reuse similar Detailed Design	GrabBag,Browsers
Implementation	Find and reuse existing routines from object code library	None
	Find and reuse (modified) source code from previous systems	GrabBag,Browsers None
	Produce Glue Code	None
Testing	Testing	Some help by Browsers
System Integration	System Integration	Some help by Browsers
Maintenance	Reuse of original product	GrabBag, Browsers, None

Table 4.1: Reusability Life Cycle Stages vs. Traditional Life Cycle

system design specification is the period of time during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirement; detailed design specification is the period of time during which the design of system or a system component is documented; typical contents include system or component algorithms, control logic, data structures, data set-use information, input/output formats, and interface description; implementation is the period of time during which a software product is created from design documentation and debugged; testing is the period of time during which the components of a software product are evaluated and integrated to determine whether or not requirements have been satisfied; system integration is the period of time during which a software product is integrated into its operational environment

and tested in this environment to ensure that it performs as required; maintenance is the period of time during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements

A component is a basic part of a system or program; an interface is a shared boundary to interact or communicate with another system component [Sta83a]. Glue code is the minimal extra code that may be needed to bring the reused modules together.

4.2 Artimis System Components

Artimis has two logically separate components: 1) GrabBag, for adding, deleting and searching for a module and its different attributes, and 2) Browsers, to aid the programmer in reading, inspecting, and understanding the code retrieved from GrabBag (or any other source of program modules).

4.2.1 GrabBag

In order to reuse existing software there must be a convenient way of locating the viable candidates for reuse. GrabBag is an indexing and retrieval system for finding available modules and their attributes in a Programmers DataBase, (PDB). PDB contains a set of *option lists* that allows the searcher to successively refine the description of the code he is looking for. Option lists are sets of *categories.* Categories are text prompts entered by the PDB builder and are used to lead the searcher to a desired module through a *search path.* A search path is a series of individual categories that lead to a module. Since there are many different ways to describe a module, there could be several different search paths to each module and it's attributes. Figure 4.1 shows a typical hierarchy of components of a PDB and possible search paths to individual module attributes. In Figure 4.1 there are

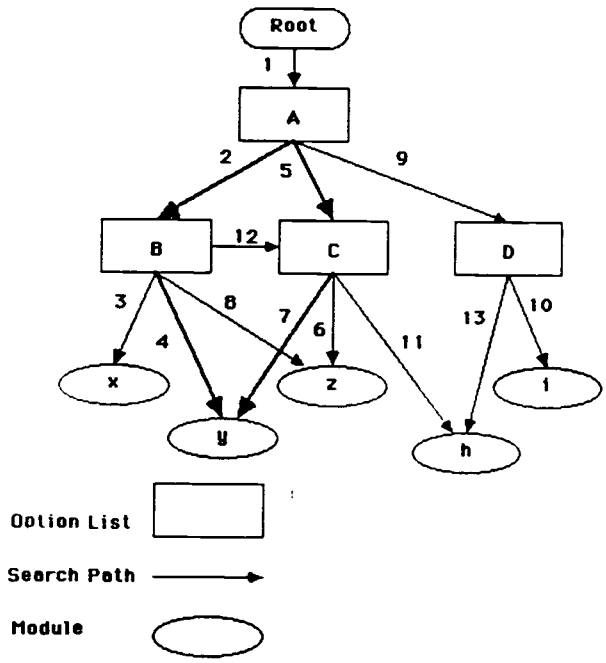


Figure 4.1: GrabBag Internal Data Model

two levels of option lists. The **Root** is a pseudo starting point of a PDB. The first option list, A, has three categories: B, C, D. Option lists B, C, and D point to some attribute files.

GrabBag Operations

GrabBag supports:

- Creation of new PDBs,
- Searching for a module and its attributes,
- Adding new Categories,
- Addition and deletion of search paths among the categories, and between categories and attribute files,

- Addition and deletion of attribute files and references to them.

The following section is a walk through and explanation of: 1) searching through a PDB to locate a category, leading to a module and its attributes, 2) adding an attribute to a module stored under an existing category, and 3) establishing a search path to a module attribute. We assume that the PDB is already selected and opened.

Searching

Once the PDB is opened GrabBag creates two windows: 1) for the display of search paths being selected in the course of searching, and 2) for display of available categories and attribute files for selection. At the beginning the title in the second window is the name of the currently opened PDB, *UTILITIES DATA BASE*, see Figure 4.2.

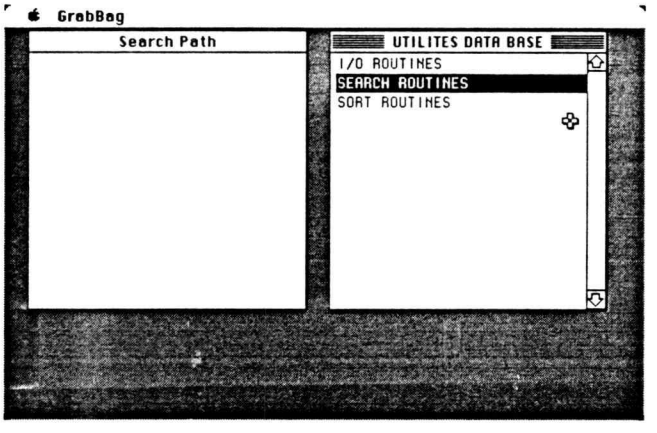


Figure 4.2: A Category and its Subcategories in a PDB

Selection of a subcategory is made by pointing to the title of the subcategory and clicking the mouse twice. In Figure 4.3 subcategory *SEARCH ROUTINES* is selected. Each time a selection is made the title of the currently selected category is

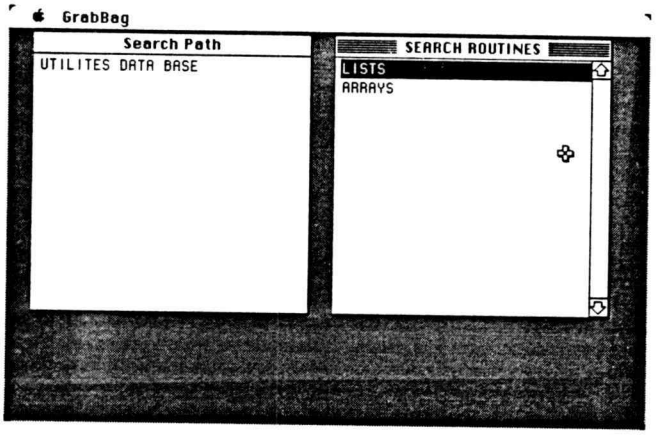


Figure 4.3: Search Path and Category Windows after a selection

updated and moved to the *Search Path* window. One can continue navigation along selected paths to narrow down choices until the desired element is found. Notice if one decides to reverse a selection and backtrack to some earlier category it is only necessary to select the category name from the *Search Path* window. Selection of a category from the *Search Path* window will always make the category the current category. This process can be repeated as long as categories exist. Reusable modules and their attribute files are stored at the end of each Search Path. Once a Search Path is exhausted, attribute file names are displayed in the left–side window as the members of the latest category. In addition, a selection dialog showing available operations is displayed as shown in Figure 4.4. The *Search Path* leading to attributes for Binary Search and the dialog box containing the available operations is shown in Figure 4.4. Selecting *Edit* will create an edit window and display the contents of the selected attribute file (List Binary S.Document) for editing or any other operations that are supported by the editor. Selection of *Copy to* will make a duplicate copy of the file; *Delete* deletes the selected attribute file from the category that it belongs to; and selecting *Cancel* removes the dialog so the search can be resumed.

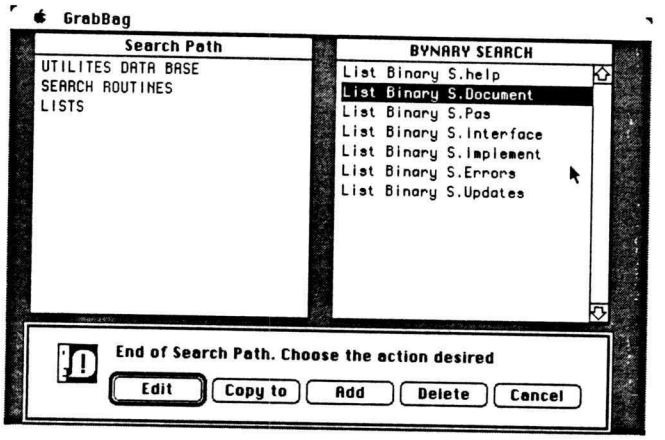


Figure 4.4: Attribute File Selection for Category

Adding New Categories

To add a new category to an option list, one first locates the desired option list (the process of locating is the same as searching for a category). Once the desired category is located, *Add Category* must be selected from the GrabBag menu shown in Figure 4.5. When *Add Category* is selected a dialog showing the category and number of subcategories already under it is displayed see Figure 4.6. The new category title is entered in the *New Subcategory* field. Selection of *Add and Quit* will add the new category as a new subcategory and quits. If there is more than one subcategory to be added, select *Keep Adding* which does the operation of adding and keeps the dialog box for further addition. Notice that the current number of subcategories under a category is also displayed. Selection of *Quit* terminates the process of adding new subcategories and returns to the category and Search Path windows.

Adding A Module Attribute

Adding a module attribute follows the same procedure for narrowing down the category by selection of categories and subcategories. Once the desired category

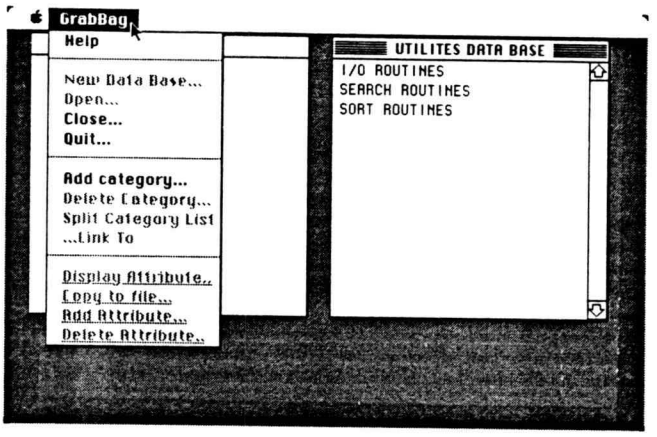


Figure 4.5: Menu item for Add Category

is located the selection of *Add Attribute* from the menu will display the name of the attribute files that can be added, see Figure 4.7, and 4.8. Selection of any of these file attributes will add them to the list of available attributes of the selected module.

Module Attribute Deletion

To delete a module, locate the subcategory which contains the module attribute to be deleted, then select *Delete Attribute* from the menu, see Figure 4.9. A dialog box will appear as shown in Figure 4.10 which tells the number of references made to the attribute file. The number of references to the specific attribute file is always shown in order to give some clue to how many active references are to that specific attribute file. One can choose to delete only the reference to the attribute file from the most recent category, or choose to delete all the references to the attribute file. In either case, the actual attribute file may be removed from the PDB by selecting the *Delete Attribute File,too* option

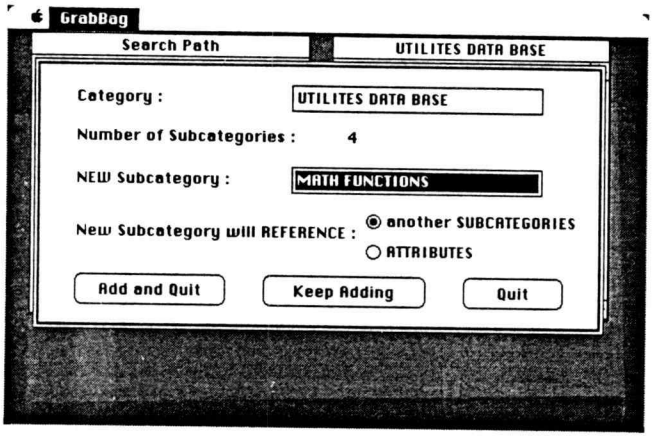


Figure 4.6: Dialog Box for Adding a New category

Linking Search Paths Among Categories and Module Attribute Files

To establish a link between a category and another category or a module attribute, the title of the *From* category must be selected from the *Search Path* window. Then the *Link From...* item must be selected from the menu to mark the category as the origin of the link, see Figures 4.11 and 4.12. Next, the module attribute or the category to which the link should point must be selected. Choosing the *Link To* from the menu specifies the destination of the link. The dialog shown in Figure 4.13 will be displayed showing what is linked to what, confirming the action. If the user decides to establish the link the *Ok* button should be pushed, otherwise the *Cancel* button should be selected.

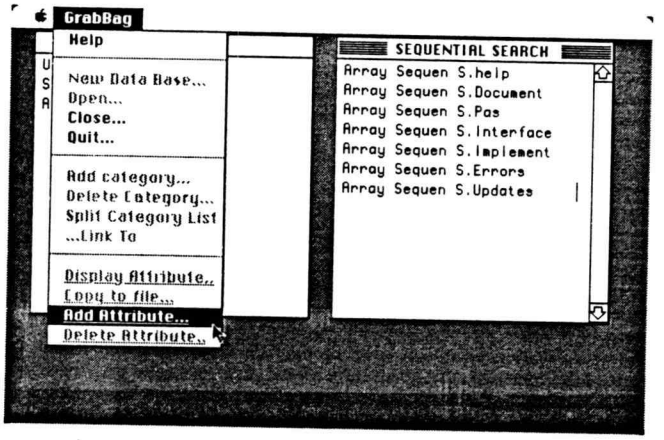


Figure 4.7: Menu Item Add Attribute Selection

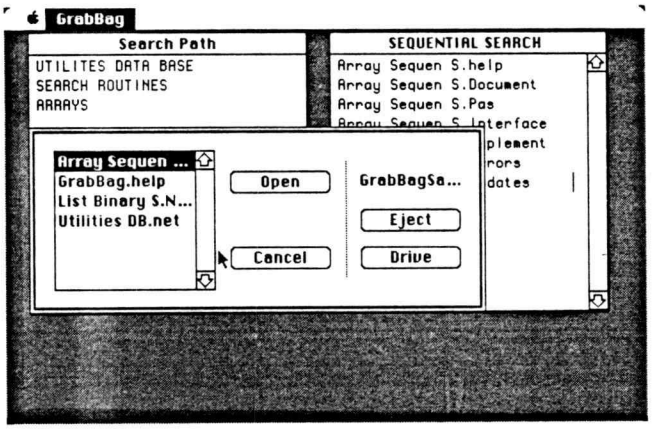


Figure 4.8: Selection Attribute File For Addition

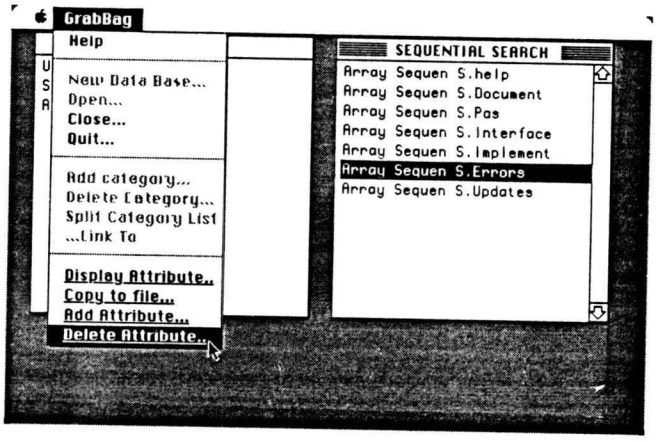


Figure 4.9: Attribute Deletion Menu Item

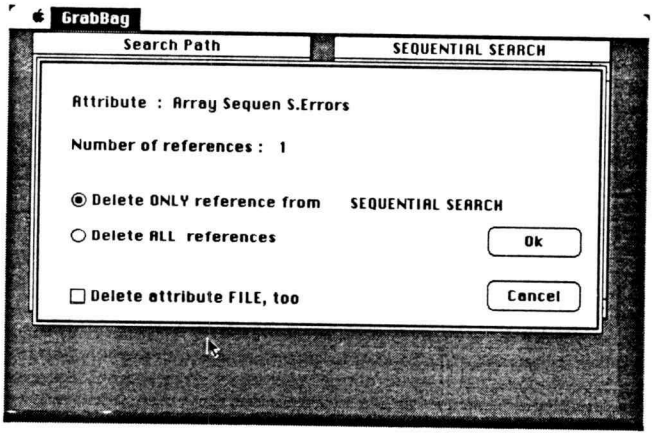


Figure 4.10: Selection Dialog For Deleting an Attribute

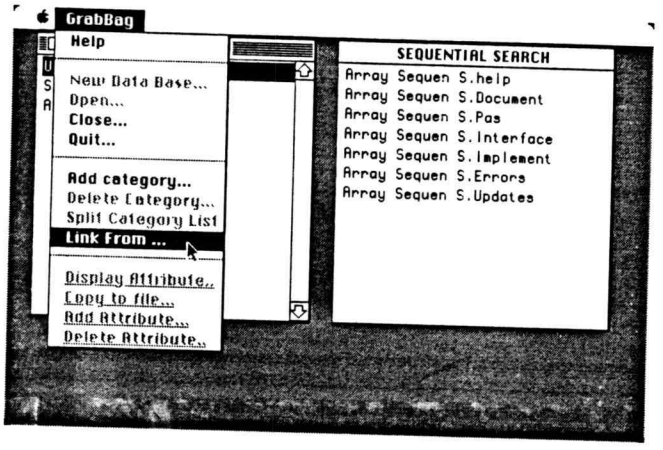


Figure 4.11: Selection of Category as the origin of the Link

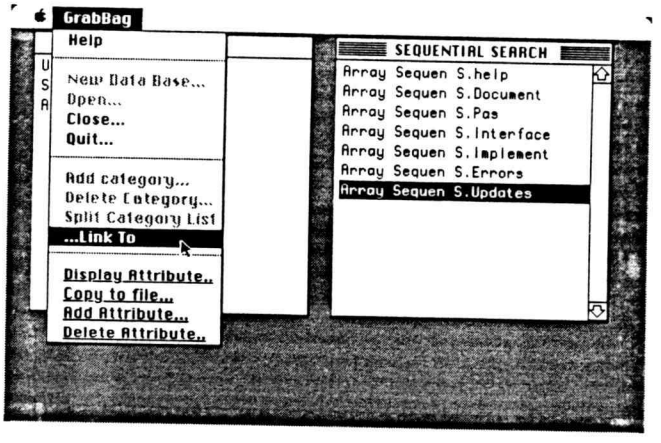


Figure 4.12: Setting the Link Origin

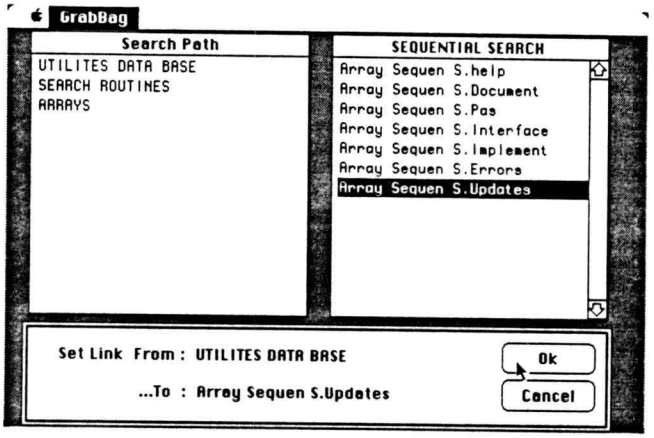


Figure 4.13: Dialog For Link Conformation

4.2.2 Browsers

Program Understanding

Browsers are tools to aid in reading and understanding program modules, a module, or parts of a module. Browsers assist the programmer in the process of mental transformation of a system of modules, a module, or parts of a module into an abstraction that summarizes the possible outcomes of the entity under consideration, irrespective of its' internal control structure and data operations.

Recent research in text comprehension [Bar32,SA77,Gra81,BBT79] has shown that schemas can facilitate the processing and storage of information by providing background knowledge or context.

> *Schemas are generic knowledge structures that guide the comprehender's interpretations, inference, expectations, and attention when passages are comprehended [Gra81]*

There is some empirical evidence [Shn76,Ade81,MRRH81] that programmers use schemas in the comprehension of computer programs. Information about the problem, *what it is* , the subgoals necessary to resolve the final goal, the method employed to solve the subgoals, *how it is done* , the level of expertise of the problem solver, etc. can be derived from program text [SE83,SEB82]. Program fragments and data structures can be thought of as schemas and knowledge structures. A program fragment is a piece of source code representing the stereotypic action sequence in programs. Program fragments are different from subroutines. Program fragments are open pieces of source code that are meant to be modified or tuned to the particular task at hand whereas subroutines are purposedly closed entities [SE83]. For example a *while* loop in a sort routine can be considered as a Loop fragment.

For a program to be reused one should know *what it is* and *how it works*. In fact understanding a source program is the basis for: 1) modifying and validating

programs written by others, 2) selecting and adapting program design, 3) verifying the correctness of programs, and 4) becoming more effective through study of programs written by others [LMW79].

Abstraction in Reading and Understanding a Module

The object of reading a program or program part is to recognize directly what it does all in one thought, or to mentally transform it into an abstraction that summarizes the possible outcomes of the program under construction irrespective of its internal control structure and data operations. Thus one can regard program reading as primarily a search for suitable abstraction [LMW79]

In [LMW79] it is shown that a program fragment is an ideal component for abstraction. A compound program of any size can be read and understood by reading and understanding its hierarchy of fragments and their abstraction. Artimis uses the idea of *stepwise abstraction* in producing an abstracted version of a module or parts of a module. The process of stepwise abstraction starts at the most detailed level, and replaces each fragment by its equivalent abstraction. Stepwise abstraction is the inverse of stepwise refinement.

4.2.3 Program Understanding Paradigm

The exact approach and steps taken in reading and understanding a program source depends on the level of expertise of the programmer, clarity and readability of the source code, and availability of documentation. The most common steps typically taken to understand a program are:

1. Build a picture of the system structure, exposing the hierarchy of interconnecting modules,

2. Examine the interface information to understand the nature and type of information exchanged among the components communicating with each other (e.g. procedures, functions, modules),
3. Start from the main program and trace the execution of the program,
4. Abstract and highlight the program fragments that are crucial to the operation of the program,
5. Comment the highlights and make notes on their operation for later use,
6. Repeat this process until the mystery is solved.

The following is the tool set which implements the steps outlined above in the Browsers of Artimis.

Module Interconnection Graph Builder

The Module Interconnection Graph Builder provides a graphical display of the hierarchical structure of a program containing one or more modules. The graphical display shows, 1) the overall program structure and placement of modules, 2) accessibility of the resources of each module from other modules, and 3) the interconnectivity (or disconnectivity) of modules. Figure 4.14 displays the Module Interconnection Graph of a set of modules.

The Module interconnection graph is the first order of fragmentation in program understanding. It provides a global view of the modules (fragments) that the program is build around. For example, Figure 4.14 (MODULES window) shows that module MOD1 has direct access to the resources (variables, procedure definitions, constants, etc.) of MOD2, and MOD3, and possibly has indirect access to the resources of MOD4, and MOD5. In turn MOD2 uses some of the resources defined in MOD4, and MOD5. These information can be used to trace the data and control flow of the module.

The module interconnection graph is also useful in formulating the dependency preserving sequence for correct compilation of the modules(MAKE). For example, in Figure 4.14 MOD4, and MOD5 should be compiled prior to MOD2 in order to preserve the correct compilation sequence.

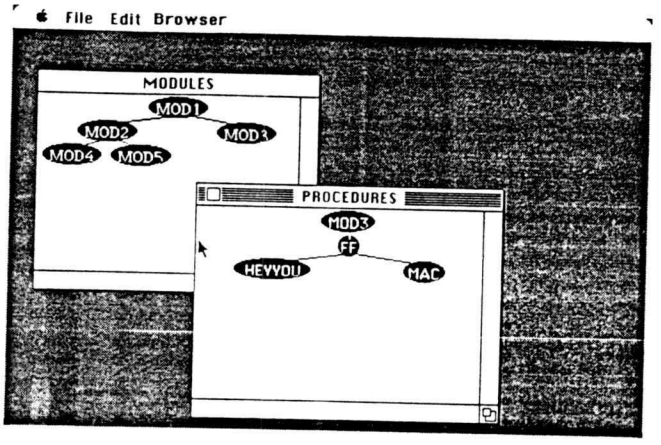


Figure 4.14: A Sample Module Interconnection and Procedure Call Graph

Procedure Call Graph Builder

The Procedure Call Graph Builder shows the subprogram invocations found within a single module. The procedure call graph is the second order of fragmentation in abstraction of a module for readability and understanding purposes. The procedure call graph reveals the textual nested organization of a module [CWW80] that can be used to derive information related to the visibility and scope of entities within a module. It provides an abstract view of the control and data flow among the subprograms. For example Figure 4.14 (PROCEDURES window) shows call graph of MOD3 in which procedure *HEYYOU* and *MAC* are called from within procedure *FF*. And *FF* is called from within body of MOD3.

Module Abstractor

The Module Abstractor automatically creates an abstraction of code fragments. This tool provides a mechanism for abstracting source code by hiding redundant and unnecessary portions of the code. The programmer can select one or many source fragments for abstraction. The selected source is hidden from view and replaced by either a note provided by the programmer or a default note provided by the abstractor. The abstracted portions of the code can be reversed.

Figure 4.15, and 4.16 display a sample program before, and after abstraction of two fragments of the code. In the example the body of *FOR* loop fragment is selected for abstraction.

The selected fragment is replaced by either a default place holder or by a prompt that is supplied by the user. For example Figure 4.15 shows the body of the *WHILE* loop is selected for abstraction. After the selection the body of the *WHILE* is hidden and replaced by *statement(s)* as shown in Figure 4.16.

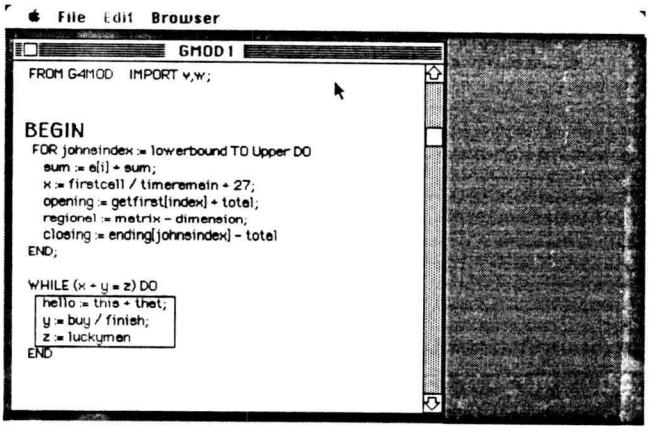


Figure 4.15: A Sample Module Before Abstraction

In Artimis, the Module Abstractor can also be used to create internal documentation. *Internal* documentation is the explanation of the algorithmic behavior of

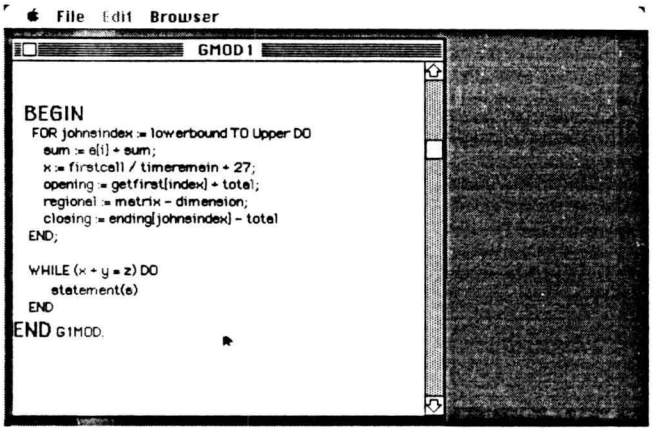


Figure 4.16: A Sample Module After Abstraction

the fragments of the code in the module. When abstracting fragments, the user can enter any annotation regarding the fragment to be abstracted. These annotations replace the actual code. This provides the capability of generating internal documentation by enabling one to produce documentation consisting of a mixture of source and annotation, annotation only, or source only.

4.3 Conclusion

It is easier to reuse program fragments than to reinvent them, provided that the time needed for program understanding is less than the time needed for program writing, and provided that the access time for the needed program fragment is sufficiently small. If these two conditions are met then the total programming and debugging time is reduced.

The GrabBag and Browser tools in Artimis provide simple, yet efficient facilities in meeting the above conditions. GrabBag's user interface provides a simple and natural method of searching and locating viable candidates for reuse. The ease of backtracking to previous search selections, and the ability to view all the available

categories in one glance creates a highly friendly, and easy environment for locating desired modules. The user interface of GrabBag also makes learning and using it so simple that one does not need to know much about it in order to use it.

The program understanding paradigm was used as a guideline in building tools that are applied to source modules in a non-intrusive manner. The Module interconnection graph and procedure call graph provide a road map and global view of the architecture of a module. The Module Abstractor further hides the non-essentials of the source program and helps to further narrow attention to portions that are essential in understanding the source. Annotating fragments while abstracting them is a natural way to retain the knowledge related to program fragments for further reuse.

4.4 References

[Ade81] B. Adelson. Problem Solving And The Development Of Abstract Categories In Programming Languages. *Memory and Cognition*, 9:422–433, 1981.

[Bar32] F.C. Bartlett. *Remembering.* University Press, Cambridge, 1932.

[BBT79] G.H. Bower, J.B. Black, and T. Turner. Scripts In Memory For Text. *Cognitive Psychology*, 11:177–220, 1979.

[Bir86] Abbas Birjandi. *A Rule Based Approach To Program Development.* PhD thesis, Computer Science Department, Oregon State University, 1986.

[Che83] T.E. Cheatham. Reusability Through Program Transformations. In *Proceedings of Workshop on Reusability in Programming*, pages 122–128, The Media Works, Inc., Newport, RI, September 1983.

[CWW80] Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf. Nesting In Ada Programs Is For The Birds. In *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*, pages 139–145, ACM, Boston, Massachusetts, December 1980.

[Fre83] Peter Freeman. Reusable Software Engineering:Concepts And Research Directions. In *Proceedings on Workshop on Reusability in Programming*, pages 2–16, Newport, 9 1983.

[Gra81] A.C. Graesser. *Prose Comprehension Beyond The Word.* Springer-Verlag, New York, 1981.

[Ker83] Kernighan. The Unix System And Software Reusability. In *Proceedings of the Workshop on Reusability in Programming*, pages 235–239, The

Media Works, Inc., Newport, RI, September 1983.

[LMW79] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming, Theory and Practice.* Addison Wesley, 1979.

[MRRH81] K.B. McKeithen, J.S. Reitman, H.H. Rueter, and S.C. Hirtle. Knowledge Organization And Skill Differences In Computer Programmers. *Cognitive Psychology*, 13:307–325, 1981.

[SA77] R.C. Schank and R. Abelson. *Scripts, Plans, Goals and Understanding.* Technical Report, Lawernce Erlbaum Associates, Hillsdale New Jersey, 1977.

[San86] Jorge Sanchez. *GrabBag:A Module Data Database.* Master's thesis, Computer Science Department, Oregon State University, 1986.

[SE83] Elliot Soloway and Kate Ehrlich. What Do Programmers Reuse? Theory And Experiment. In *Proceedings of Workshop on Reusability in Programming*, pages 184–191, The Media Works,Inc., Newport, RI, September 1983.

[SEB82] E. Soloway, K. Ehrlich, and J. Bonar. Tapping Into Tacit Programming Knowledge. In *Proceedings of the Conference on Human Factors in Computing Systems*, NBS, Gaithersburg, Md., 1982.

[Shn76] B. Shneiderman. Exploratory Experiments In Programmer Behavior. *International Journal of Computer and Information Sciences*, 5:123–143, 1976.

[Sta83] American National Standard. *IEEE Standard Glossary of Software Engineering Terminology.* New York, February 1983.

[Wir83] Niklaus Wirth. *Programming In Modula-2. Texts and Monographs in Computer Science*, Springer-Verlag, Berlin Heidelberg, 1983.

Chapter 5

Bibliography

[Ade81] B. Adelson. Problem Solving And The Development Of Abstract Categories In Programming Languages. *Memory and Cognition*, 9:422–433, 1981.

[Bal83] Robert Balzer. Evolution As A New Basis For Reusability. In *Proceedings of Workshop on Reusability in Programming*, pages 80–82, The Media Works, Inc., Newport, RI, September 1983.

[Bar32] F.C. Bartlett. *Remembering.* University Press, Cambridge, 1932.

[Bar79] David Barstow. *Knowledge-Based Program Construction.* Elsevier North Holand, 1979.

[BBT79] G.H. Bower, J.B. Black, and T. Turner. Scripts In Memory For Text. *Cognitive Psychology*, 11:177–220, 1979.

[BGW76] R. Balzer, N. Goldman, and D. Wile. On The Transformational Implementation Approach To Programming. In *Proceedings of the Second International Conference on Software Engineering*, IEEE Computer Society, Long Beach, Calif., 1976.

[Bir82] Abbas Birjandi. *A Programming Environment For Creation And Maintenance Of Reusable Software Modules.* PhD thesis proposal, Oregon State University, Corvallis, 1982.

[Bir86] Abbas Birjandi. *A Rule Based Approach To Program Development.* PhD thesis, Computer Science Department, Oregon State University, 1986.

[BL86a] Abbas Birjandi and T. G. Lewis. *Arash:A Re-Structuring Tool For Building Software Systems From Reusable Components.* Technical Report 86-10-2, Department of Computer Science Oregon State University, Corvallis Oregon 97331, 1986.

[BL86b] Abbas Birjandi and T. G. Lewis. *Artimis: A Module Indexing And Source Program Reading And Understanding Environment.* Technical Report 86-10-3, Department of Computer Science Oregon State University, Corvallis Oregon 97331, 1986.

[BL86c] Abbas Birjandi and T. G. Lewis. *Yashar:A Rule Based Meta-Tool For Program Development.* Technical Report 86-30-6, Department of Computer Science Oregon State University, Corvallis Oregon 97331, 1986.

[Bro72] P. J. Brown. Re-creation Of Source Code From Reverse Polish Form. *Software–Practice and Experience*, 2:275–278, 1972.

[Bro77] P. J. Brown. More On The Re-creation Of Source Code From Reverse Polish Form. *Software–Practice and Experience*, 7:545–551, 1977.

[Bro82] P. J. Brown. *Tools For Amateurs. Tools And Notations for Program Construction, An Advanced Course*, Cambridge University Press, 1982.

[CH73] C. C. Charlton and P. G. Hibbard. A Mote On Recreating Source Code From The Reverse Polish Form. *Software–Practice and Experience*, 3:151–153, 1973.

[Che83] T.E. Cheatham. Reusability Through Program Transformations. In *Proceedings of Workshop on Reusability in Programming*, pages 122–128, The Media Works, Inc., Newport, RI, September 1983.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog. Texts and Monographs in Computer Science*, Springer-Verlag, New York, 2 edition, 1984.

[Col85] Alain Colmerauer. Prolog In 10 Figures. *Communication Of The ACM*, 28:1296–1324, December 1985.

[CP83] C.S. Chandersekaran and M.P. Perriens. Towards An Assessment Of Software Reusability. In *Proceedings of Workshop on Reusability in Programming*, pages 179–182, The Media Works, Inc., Newport, RI, September 1983.

[CS73] W.C. Chase and H. Simon. Perception in Chess. *Cognitive Psychology*, 4:55–81, 1973.

[Cur83] Bill Curtis. Cognitive Issues In Reusability. In *Proceedings of Workshop on Reusability in Programming*, pages 192–197, The Media Works, Inc., Newport, RI, September 1983.

[CWW80] Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf. Nesting In Ada Programs Is For The Birds. In *Proceedings of The ACM-SIGPLAN Symposium on the Ada Programming Language*, pages 139–145, ACM, Boston, Massachusetts, December 1980.

[deG65] A.D. deGroot. *Thought and Choice in Chess.* Mouton and Company, Paris, 1965.

[Deu83] L. Peter Deutsch. Reusability In The Smalltalk-80 Programming System. In *Proceedings of Workshop on Reusability in Programming*, pages 72–76, The Media Works, Inc., Newport, RI, September 1983.

[DIA83] DIANA. *DIANA:An Intermediate Language for ADA Revised Version.* Volume 161 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1983.

[DK76] F. DeRemer and H. H. Kron. Programming-In-The-Large Versus Programming-In-The-Small. *IEEE Trans. Software Eng.*, SE-2:80–86, 1976.

[DKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Melese. Document Structure And Modularity In Mentor. In *Proceedings of The ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 141–148, ACM, Pittsburgh,Pennsylvania, April 1984.

[DOD82] DOD. *Reference Manual For The Ada Programming Language.* United States Department of Defense, Washington DC, July 1982.

[DVHK80] V. Donzeau-Gouge, Veronique, Huet, and G. Kahn. Programming Environment Based On Structured Editors:The Mentor Experience. In *Workshop on Programming Environments*, Ridgefield, CT, June 1980.

[EGK84] J. Estublier, S. Ghoul, and S. Krakowiak. Preliminary Experience With A Configuration Control System For Modular Programs. In *Proceedings of The ACM SIGSOFT/SIGPLAN Software Engineering Symposium on*

Practical Software Development Environments, pages 1149–156, ACM, Pittsburgh,Pennsylvania, April 1984.

[EP84] V. B. Erickson and J. F. Pellegrin. Build–A Software Construction Tool. *AT&T Bell Laboratories Technical Journal*, 63:1049–1059, 7-8 1984.

[ES83] K. Ehrlich and E. Soloway. An Empirical Investigation Of The Tacit Plan Knowledge In Programming. *Human Factors in Computer Systems*, 1983?

[Fea83] Martin S. Feather. Reuse In The Context Of A Transformation Based Methodology. In *Proceedings of Workshop on Reusability in Programming*, pages 50–58, The Media Works, Inc., Newport,RI, September 1983.

[Fel79] S. I. Feldman. Make–A Program For Maintaining Computer Programs. *Software Practice and Experience*, 9(4):255–266, 1979.

[FM81] Peter H. Feiler and Raul Medina-Mora. An Incremental Programming Environment. *IEEE Trans. Software Eng.*, SE-7(5):44–53, 1981.

[FO82] Hohichi Futatsugi and Koji Okada. A Hierarchical Structuring Method For Functional Software Systems. In *Proceedings of Sixth International Conference on Software Engineering*, September 1982.

[Fre83] Peter Freeman. Reusable Software Engineering:Concepts And Research Directions. In *Proceedings on Workshop on Reusability in Programming*, pages 2–16, Newport, 9 1983.

[Fri83] Peter Fritzson. *Adaptive Prettyprinting of Abstract Syntax Applied to ADA and PASCAL.* Technical Report, Department of Computer Science, Linkoping University, Linkoping, Sweden, September 1983.

[Fri84] Peter Fritzson. *Towards A Distributed Programming Environment Based on Incremental Compilation.* PhD thesis, Department of Computer and Information Science, Linkoping Sweden, 1984.

[Gea82] C. Green and et al. *Research on Knowledge-Based Programming and Algorithm Design.* Technical Report, Kestrel Institute, Palo Alto, 1982.

[Ger83] Susan L. Gerhart. Reusability Lessons From Verification Technology. In *Proceedings of Workshop on Reusability in Programming*, pages 110–121, The Media Works,Inc., Newport, RI, September 1983 1983.

[Gog83] Joseph Goguen. Parameterized Programming. In *Proceedings of Workshop on Reusability in Programming*, pages 138–150, The Media Works, Inc., Newport, RI, September 1983.

[Goo81] James W. Goodwin. Why Programming Environments Need Dynamic Data Types. *IEEE Trans. Software Eng.*, SE-7(5):451–457, 1981.

[Gra81] A.C. Graesser. *Prose Comprehension Beyond The Word.* Springer-Verlag, New York, 1981.

[GWS81] Steve Guts, Anthony I. Wasserman, and Michael J. Spier. Personal Development Systems For The Professional Programmer. *IEEE Computer*, 14(4):45–53, 1981.

[Hab82] A. N. Habermann. *System Development Environments. Tools and Notations for Program Construction, An Advanced Course*, Cambridge University Press, 1982.

[Hal77] M.M. Halstead. *Elements Of Software Science.* Elsevier North Holland, New York, 1977.

[Han79] David R. Hanson. A Simple Technique For Controlled Communication Among Separately Compiled Modules. *Software-Practice and Experience*, 9(8):921–924, 1979.

[How81] William Howden. Contemporary Software Development Environments. *ACM SIGSOFT Software Engineering Notes*, 6:6–14, 1981.

[Joh75] Stephen C. Johnson. *Yet Another Compiler-Compiler.* Technical Report, Bell Laboratories, Murray Hill, N.J., 1975.

[Jon83] Capers Jones. Reusability In Programming: A Survey Of The State Of The Art. In *Proceedings of Workshop on Reusability in Programming*, pages 215–222, The Media Works, Inc., Newport, RI, September 1983.

[JW76] K. Jensen and Niklaus Wirth. *Pascal User Manual and Report. Texts and Monographs in Computer Science*, Springer-Verlag, 2 edition, 1976.

[Kan85] Kirk Kandt. Pegasus:A Software Design Tool. In *Proceedings of The 18th Annual Hawaii International Conference on System Sciences*, pages 650–657, University of California, Irvine, California, 1985.

[Ker83] Kernighan. The Unix System And Software Reusability. In *Proceedings of The Workshop on Reusability in Programming*, pages 235–239, The Media Works, Inc., Newport, RI, September 1983.

[KM81] Brian W. Keringhan and John R. Mashey. The Unix Programming Environment. *IEEE Computer Magazine*, 14(4):12–24, 1981.

[Knu73] Donald E. Knuth. *The Art of Computer Programming.* Volume 1, Addison Wesley, 2 edition, 1973.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language. Prentice-Hall Software Series*, Prentice-Hall, 1978.

[KS83] Gary D. Kimura and Alan C. Shaw. *The Structure Of Abstract Document Objects.* Technical Report, Computer Science Dept. University of Washington, Seattle, Washington, September 1983.

[Led83] Lamar Ledbetter. Reusability Of Domain Knowledge In The Automatic Programming System. In *Proceedings of Workshop on Reusability in Programming*, pages 97–105, The Media Works, Inc., Newport,RI, September 1983.

[Lin84] Mark A. Linton. Implementing Relational Views Of Programs. In *Proceedings of The ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132–140, ACM, Pittsburgh,Pennsylvania, April 1984.

[Lis72] B. H. Liskov. A Design Methodology For Reliable Software Systems. In *Fall Joint Computing Conference*, pages 191–199, AFIPS Press, Montvale, NJ, 1972.

[LM75] Henry F. Legard and Michael Marcotty. A Genealogy Of Control Structures. *CACM*, 18:629–639, 11 1975.

[LM83] S.D. Litvintchouk and A.S. Matsumoto. Design Of Ada Systems Providing Reusable Components. In *Proceedings of Workshop on Reusability in Programming*, pages 198–206, The Media Works, Inc., Newport, RI, September 1983.

[LMW79] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming, Theory and Practice.* Addison Wesley, 1979.

[LN85] C. Lewerentz and M. Nagi. Incremental Programming In The large:Syntax-Aided Specification editing, integration and maintenance. In *Proceedings of The 18th Annual Hawaii International Conference on*

System Sciences, pages 638–649, Angewandte Informatik, FB Mathematik AND Informatic Universitaet Osnabrueck, Osnabrueck, 1985.

[Mac83] Bruce J. MacLennan. *Principle of Programming Languages:Design, Evaluation, and Implementation.* CBS College Publishing, 1983.

[MB84] N. Minsky and A. Borgida. The Darwin Software-Evaluation Environment. In *Proceedings of The ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 89–95, ACM, Pittsburgh,Pennsylvania, April 1984.

[Mee78] L. Meertens. Program Text And Program Structure Constructing Quality Software. *IFIP, North-Holland Publishing Co.*, 1978.

[Mil70] Harlan D. Mills. Syntax-Directed Documentation For PL360. *Communication of The ACM*, 13(4):216–222, April 1970.

[MRRH81] K.B. McKeithen, J.S. Reitman, H.H. Rueter, and S.C. Hirtle. Knowledge Organization And Skill Differences In Computer Programmers. *Cognitive Psychology*, 13:307–325, 1981.

[MVL85] Nazim H. Madhavji, Dimitri Voulioutis, and Nikos Leoutsarakos. The Importance of Context In An Integrated Programming Environment. In *Proceedings of The 18th Annual Hawaii International Conference on System Sciences*, pages 608–624, School of Computer Science McGill University, Montreal, CANADA, 1985.

[Nei83] James M. Neighbors. The Draco Approach To Constructing Software From Reusable Components. In *Proceedings of Workshop on Reusability in Programming*, pages 167–177, The Media Works, Inc., September 1983.

[NH81] David S. Notkin and Nico Habermann. Software Development Environment Issues As Related To Ada. In *Tutorial:Software Development Environments*, pages 107–137, IEEE Computer Society, 1981.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph In A Software Development Environment. In *Proceedings of The ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, ACM, Pittsburgh,Pennsylvania, April 1984.

[Opp80] D. Oppen. Prettyprinting. *ACM Transaction on Program Languages and Systems*, 2(4), October 1980.

[Ost81] Leon Osterweil. Software Environment Research: Directions For The Next Five Years. *IEEE Computer Magazine*, 14(4):35–43, 1981.

[PLS82] Luigi Petrone, Antonio Di Leva, and Franco Sirovich. Dual: An Interactive Tool For Developing Documented Programs By Step-wise Refinments. In *Proceedings of Sixth International Conference on Software Engineering*, September 1982.

[Pra75] Terrance W. Pratt. *Programming Languages: Design and Implementation.* Prentice-Hall, Englewood Cliffs, N. J., 1975.

[PS83] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, 15:199–236, September 1983.

[RCC82] A. Rudmik, B. E. Casey, and H. Cohen. Consistency Checking Within Embedded Design Languages. In *Proceedings of Sixth International Conference on Software Engineering*, September 1982.

[Rei85] Steven P. Reiss. Program Development Systems That Supports Multiple Views. *IEEE Transaction on Software Engineering*, SE-11(3), March 1985.

[Ric81] C. Rich. *Inspection Methods in Programming.* Technical Report, MIT AI Lab, 1981.

[Roc75] Marc J. Rochkind. The Source Code Control System. *IEEE Trans. Software Eng.*, SE-1, 1975.

[RS82] J. Ramanathan and C. Shubra. Use of Annotated Schemes For Developing Prototype Programs. *ACM SIGSOFT Software Engineering Notes*, 7(5):141–149, 1982.

[RS83] John R. Rice and Herbert D. Schwetman. Interface Issues in A Software Parts Technology. In *Proceedings of Workshop on Reusability in Programming*, pages 129–137, The Media Works, Inc., Newport, RI, September 1983.

[RT85] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Technical Report, Department of Computer Science Cornell University, Ithaca NY 14583, 8 1985.

[RW83] Charles Rich and Richard C. Waters. Formalizing Reusable Software Components. In *Proceedings of Workshop on Reusability in Programming*, pages 152–159, The Media Works, Inc., Newport, RI, September 1983.

[SA77] R.C. Schank and R. Abelson. *Scripts, Plans, Goals and Understanding.* Technical Report, Lawerance Erlbaum Associates, Hillsdale New Jersey, 1977.

[SAN*81] Mary Shaw, Guy T. Alems, Joseph M. Newcomer, Brian K. Reid, and William A. Wulf. Programming Language For Software Engineering. *Software Practice and Experience*, 11:1–52, 1981.

[San86] Jorge Sanchez. *GrabBag:A Module Data Database.* Master's thesis, Computer Science Department, Oregon State University, 1986.

[SBE82] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive Strategies And Looping Constructs:An Empirical Study. *CACM*, 1982.

[SE83] Elliot Soloway and Kate Ehrlich. What Do Programmers Reuse? Theory And Experiment. In *Proceedings of Workshop on Reusability in Programming*, pages 184–191, The Media Works,Inc., Newport, RI, September 1983.

[SEB82] E. Soloway, K. Ehrlich, and J. Bonar. Tapping Into Tacit Programming Knowledge. In *Proceedings of The Conference on Human Factors in Computing Systems*, NBS, Gaithersburg, Md., 1982.

[Shn76] B. Shneiderman. Exploratory Experiments In Programmer Behavior. *International Journal of Computer and Information Sciences*, 5:123–143, 1976.

[Sta81] Thomas A. Standish. Advanced Development Support Systems. *Software Engineering Notes*, 6:25–35, August 1981.

[Sta83a] American National Standard. *IEEE Standard Glossary of Software Engineering Terminology.* New York, February 1983.

[Sta83b] Thomas A. Standish. Software Reuse. In *Proceedings of Workshop on Reusability in Programming*, pages 45–49, The Media Works, Inc., Newport, RI, September 1983.

[TM81] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. *IEEE Computer Magazine*, 14(4):25–33, 1981.

[TR80] Tim Teitelbaum and Thomas Reps. *The Cornell Program Synthesizer:A Syntax-Directed Programming Environment.* Technical Report, Computer Science Dept. Cornell University, MAY 1980.

[Tra79] William J. Tracz. Computer Programming And The Human Thought Process. *Software Practice and Experience*, 9(1):127–137, 1979.

[War82] Sally Warren. Map: A Tool For Understanding Software. In *Proceedings of Sixth International Conference on Software Engineering*, September 1982.

[Weg74] Ben Wegbreit. The Treatment Of Data Types In El1. *Communication of The ACM*, 17(5):251–264, May 1974.

[Weg83] Peter Wegner. Varieties Of Reusability. In *Proceedings of Workshop on Reusability in Programming*, pages 30–44, The Media Works, Inc., Newport,RI, 9 1983.

[Wer82] Harald Wertz. The Design Of An Integrated, Interactive And Incremental Programming Environment. In *Proceedings of Sixth International Conference on Software Engineering*, September 1982.

[Wil80] J. Wilander. An Interactive Programming System For Pascal. *BIT*, 20:163–174, 1980.

[Wir83] Niklaus Wirth. *Programming In Modula-2. Texts and Monographs in Computer Science*, Springer-Verlag, Berlin Heidelberg, 1983.

[Wte81] R. Wters. A Knowledge Based Program Editor. In *Proceedings of The 7th IJCAI Conference*, 1981.

[YN85] T. Yuasa and R. Nakajima. Iota:A Modular Programming System. *Transaction On Software Engineering*, SE-11(2):179–187, 1985.

[Zel81] Marvin Zelkowitz. High Level Language Programming Environment. *ACM SIGSOFT Software Engineering Notes*, 6(4):36–43, 1981.

Appendices

Appendix A

Rule Processor Instruction Set

In the following sections the instruction set of the rule processor and their semantics are explained. Instructions are preceded by an @ to distinguish them from ordinary textual information. The mandatory portions of the commands are enclosed between { and }. The optional portions are enclosed between [and]. Also α ...α means a string of characters enclosed by two identical symbols. The symbol α can be any character. The string must not contain the symbol α. The notation of the instructions are adapted from [Fri83]. The meta symbols < and > enclose non-terminals such as arithmetic expressions.

Rule Context

To traverse the tree according to the rules stored in the rule repository, the rule processor maintains a *context* for each node. A context consists of:

Node Priority: used in generation of parenthesized expression. There exists a classical problem of regenerating expressions from expression trees when the priority of their operators is altered by using parenthesis [Bro72,CH73,Bro77]. We have adopted the solution in [Fri83] in which expression sub-trees are assigned priorities and associativity values. These values provide the capability to decide where to emit parenthesis in regeneration of expressions. Yashar's

solution is a generalization that assigns priorities to a node type rather than the expression node so that the rule processor will emit the proper parentheses. In this new method there is no need to specify the associativity relation of a node.

Delimiter String Pointer: used to replicate common strings of symbols shared by children of a node. Such delimiters are assigned to a delimiter string buffer area and emitted when they are needed.

Pointer to Rule Definition: refers to the rule definition of a node. The rule for each node to be processed is prefetched prior to its execution and its address is passed to the rule processor.

A.1 Tree Navigational Instructions

@n Process the *n*th child of the current node. If the child does not exist the rule processor ignores this instruction.

@Xn Process the tree (subtree) pointed to by the contents of register *Rn*. This command is used in conjunction with the @M, move command, see A.3. The execute instruction takes the register *n* as the current root node, register *n+1* as the current arithmetic priority, and register *n+2* as pointer to current delimiter string.

@.RT{A} Remove the subtree pointed by A where A is either a register number which points to the subtree. If the pointer is null, no action take places. Not implemented yet.

@.CT{A_1, A_2} Make a copy of the subtree pointed by A_1 and set A_2 to the address of created subtree. A_1 andA_2 must be registers only. Not implemented yet.

@.PT$\{A_1, A_2$,[S|K]n$\}$ Paste subtree pointed by A_2, to subtree pointed by A_1 as nth sibling (S) or child (K). Not implemented yet.

@In Do not process the children number n of the current node. Continues with the next children if there is any.

@A$\alpha \ldots \alpha$**@nn** Instead of processing child number nn of this node take whatever enclosed between α's as the result of processing and continue with the next instruction.

@ARn$\alpha \ldots \alpha$**@nn** Set register n to number of children of children number nn of current node. Instead of processing it return whatever is enclosed between α's as the result of processing this node.

@An$\alpha \ldots \alpha$**@nn** Instead of processing children number nn of current node take whatever is between the delimiters and append to it number of children of child number nn as the result of processing and continue with the next instruction.

@AnRn$\alpha \ldots \alpha$**@nn** Do as above but also save the number of children of children number nn of current node in register number n.

@D$\alpha \ldots \alpha$ Emit whatever is enclosed between delimiters after processing of each child of the current node that comes afterwards.

@*D After processing of each child of current node emit the same information that is defined by the most currently set **D** instruction prior to this.

@* $\alpha \ldots \alpha$ Emit whatever is enclosed between delimiters after processing of each child of the current node. Note this is only local to this node.

A.2 Escape and Break point Instructions

The rule processor supports the insertion (definition) , activation, and removal of break points to temporarily interrupt the processing of a rule. One can use the break point facility to step through a class of tree nodes, for example. Definition, activation and removal of break points are done as follows:

Definition: of a break point for a node type is done by using **@P** and providing the node type and function number in the Function Table to be activated at the time of breaking. For example:

$$05: \overbrace{@P}^{define\ break}\ \overbrace{/}^{separator}\ \underbrace{20}_{node\ type}\ \overbrace{/}^{separator}\ \underbrace{03}_{function\ to\ breakin}\ CONST@D...$$

defines a break point for all nodes of type 20 and designates the function number 3 in the Function Table as the function to be executed when a break happens.

Removal: of a break point for a node type is done by using **@V** and the specification of the node type. For example the following:

$$84: BEGIN@n@+@01...END\ \overbrace{@V}^{remove\ break}\ /\ \underbrace{20}_{node\ type}\ /$$

removes the break definition for node type 20.

Activation: of break points is done by using **@Z.** to alert the rule processor to check for a possible break point definition for the current node type. The sole purpose of **@Z** is to not hinder the efficiency of the rule processor when checking for break points after execution of each rule. However, by adding **@Z** to all rule definitions, checking for a break point after every rule can be achieved. For example ,

16:IF @01 THEN ...END @Z

38:WHILE @01 @D/;@n/ ...END @Z

50:(@01 > @02)@Z

will cause the rule processor to check for a possible break point definition for node types 16, 38, and 50 after processing them. Notice that by putting @Z at the beginning one can cause the rule processor to check for possible break points at the beginning of a rule. Practically, @Z can be placed anywhere within a rule and it's execution is immediate.

@n%m Causes the rule processor to by pass processing of child number *n* and execute function referenced at location *m* of the Function Table. The rule processor passes the *context* to the function too.

@Jn Calls *nth* function in Function Table. No *context* information is passed to the function.

@Pαnαm Defines a break point for node type *n* and designates the *m*th function in the Function Table to be the breaking function.

@Vαnα Removes the break point definition for node type *n*.

@Z Defines the check points for existence of a break point. Meaning that any time a @Z is encountered the rule processor checks to see if a break point is defined for the rule and if so executes the breaking function as it is defined in @P for the rule.

A.3 Arithmetic and Conditional Instructions

@?(<exp>)?$\overbrace{[n : newrule]}^{true\ part}$?$\underbrace{[n : newrule]}_{false\ part}$? Conditionally modifies a command depending on whether the condition being test is true or false. The <exp> is

evaluated and if the result is true the *true part* is used otherwise the *false part.*

@M$Rn =(<exp>)$ Set the contents of register *n* to the result of the expression.

Expressions can contain any combination of arithmetic operations (eg. +, - , etc.) and relational operators (eg. ==, <=, etc.) in case of conditional instructions. If the expression is a register assignment, the next two consecutive registers are used to store the current value of arithmetic priority, and the pointer to current active string delimiter. For example

$$05 : CONST@D/;@n/@n@ + @M\$R02 = (@01)\$@ * //;@n@-$$

In the above the sequence operation related to assignment is as follows:

- Store the pointer to subtree @01 in register 02
- Store the current arithmetic priority in register 03
- Store the pointer to current delimiter string which happens to be ;@n in register 04

A.4 Formatting Instructions

@+ Increment current indentation level by a predefined indentation value. The execution of this command is immediate.

@- Decrement current indentation level. This is the reverse of @+.

@C+ Enable adaptive formation of output text. This signals the rule processor to attempt to break lines of output text that does not fit on a single line.

@C- Disable adaptive formation of output text. This disables the effect of a previous @C+. Afterward the rule processor does not make any attempt to adjust the display of the text if it does not fit in a line.

@Cm This is a marker which to mark the spots that would be a reasonable place to emit a proper escape sequence (e.g newline) to break the output lines. When the adaptive formation of output text is enabled. This marker provides the knowledge to the rule processor in calculating the most appropriate places that a line can be broken.

@.F[+|-]n where **F** can be any of the following font styles: **Bold**, *Italic*, Underline, Outline, Shadow, and Normal. These font styles can be selected independent of each other and their effects are accumulative. To reset the font style to the default one one should select Normal.

@.G[+|-]n| r In this instruction The **n** option can be used to set the font size. The **r** option resets the font size to the default value, or the font size prior to the application of a set operation.

@^n Set the arithmetic priority of current node to *n*. This is for generating of parenthesized expression in a correct form when generating program text.

A.5 Miscellaneous Instructions

Rule Repository Manipulation Instructions

@Mnα ...α Redefine rule labeled *n* to the new definition enclosed in between delimiters. After the execution of this command the new definition will be in effect.

@Rn Restore the definition of rule labeled *n* to its previous one. If there is no previous definition nothing will be changed.

@F Restor all the rules that are redefined to their original definitions.

Data Dictionary Access and Pre-Loading of Modified Rules

@d The access to Data Dictionary from within rule is through the use of **@d**. However to make that possible prior to activation of rule processor a designer-defined function is installed in a predefined element of Function Table. Then afterwards anytime the rule processor encounters **@d** it will execute the installed function in the pre-defined location of the Function Table and the content of Data Dictionary Reference Pointer field of the current node is passed to it. The return value is expected to be a string of characters. However a null string can also be returned. The data dictionary access routine must always be installed as function number **41**.

Pre-Loading of Modified Rules For pre-loading modified rules before starting activation of rule processor, a designer-defined function must be installed as a pre-defined element of Function Table. Yashar rule processor always attempts to execute this function before start of processing the rules. If there is no function installed in that location nothing will happen and processing will continue. The designer-defined routine to modify or augment the designer-defined rules prior to activation of rule processor must be installed as function number **42**. Further more this function must use pre-compiled Yashar support routine *ModifyRule*, refer to Appendix B.2, to actually modify the rules.

A.6 References

[Bro72] P. J. Brown. Re-creation Of Source Code From Reverse Polish Form. *Software–Practice and Experience*, 2:275–278, 1972.

[Bro77] P. J. Brown. More On The Re-creation Of Source Code From Reverse polish Form. *Software–Practice and Experience*, 7:545–551, 1977.

[CH73] C. C. Charlton and P. G. Hibbard. A note On Recreating Source Code From The Reverse Polish Form. *Software–Practice and Experience*, 3:151–153, 1973.

[Fri83] Peter Fritzson. *Adaptive Prettyprinting of Abstract Syntax Applied to ADA and PASCAL.* Technical Report, Department of Computer Science, Linkoping University, Linkoping, Sweden, September 1983.

Appendix B

Pre-Compiled Support Routines

The *pre-compiled* support functions are used by a tool of Yashar to access and operate on the tree-representation of the input and Scratch Pad Area. The pre-compiled support functions are accessible through designer-defined functions and Application Interface Routines.

The *designer-defined* functions are application-specific routines that are called by the rule processor to perform certain application-specific tasks. User-defined support functions are installed in the rule processor's predefined *Function Table* by using a pre-compiled Yashar support routine called *InstFunc*. User-defined support functions can be viewed as trap routines which extend the instruction set of Yashar's rule processor. The rule processor executes functions installed in the Function Table automatically when they are referenced in any rule.

As an example suppose user interaction is required to satisfy one of the rules being processed by Yashar's rule processor. The designer would have to write a function which handles the user interaction as a dialogue and passes the user's input to Yashar's rule processor. Prior to activation of the rule processor this function is installed in the Function Table by using the pre-compiled Yashar support function *InstFunc*. The rule processor automatically executes such functions in the course of processing the rules.

Following sections explain the pre-compiled support routines that are available for User Interface Routines and designer-defined routine to access the tree-representation of input, the Scratch Pad area and rule repository.

B.1 Tree Manipulation Routines

Tree manipulation routines provide the necessary support for creating and accessing tree nodes.

NewTreeNode(): creates a tree node and returns a pointer to it.

GetIthSib(*treenode*, *sibNum*): returns a pointer the *sibNum*th sibling of the tree or subtree passed to it as its argument. *treenode* is the pointer pointing to the specified tree or subtree, and *sibNum* is the desired siblings. If the desired sibling does not exist the return pointer will be null.

GetIthKid(*treenode*, *kidNum*): returns a pointer the *kidNum*th child of the tree or subtree passed to it as its argument. *treenode* is the pointer pointing to the specified tree or subtree, and *kidNum* is the desired child. If the desired child does not exist the return pointer will be null.

AddSib(*treenode*,*newsibs*): adds the tree node pointed by *newsibs* as the last sibling of the tree subtree pointed by *treenode*.

AddKid(*treenode*,*newkids*): adds the tree node pointed by *newkids* as the last child of the tree pointed by *treenode*.

NoOfSibs(*treenode*): returns an integer value representing the number of the siblings of the tree pointed by *treenode*.

NoOfKids(*treenode*): returns an integer value representing the number of the children of the tree pointed by *treenode*.

CopySubTree(*treenode*): makes a copy of the tree pointed by *treenode* and returns a pointer to the newly created tree.

SetType(*treenode*,*TypeValue*): set the type of the node pointed by *treenode* to the value of *TypeValue*. Currently the *TypeValue* can only be in the range 0 to 255.

GetType(*treenode*): returns an integer as the value of type field of the tree pointed by *treenode*.

SetDRef(*treenode*,*DDRefInfo*): sets the data dictionary reference field of the node pointed by *treenode* with the content of *DDRefInfo*.

GetDRef(*treenode*): returns the contents of data dictionary reference field of the node pointed by *treenode*. The return value is four byte long and can be casted to a pointer or a long integer in C.

SetALink(*treenode*,*ALinkInfo*): sets the application linkage/data field of the node pointed by *treenode* with the contents of *ALinkInfo*.

GetALink(*treenode*): returns the contents of application linkage/data field of the node pointed by *treenode*. The return value is four byte long and can be casted to a pointer or a long integer in C.

B.2 Repository Manipulation Routines

These routines provide access mechanism to the rule repository, and the ability of modifying the default size of them. Except *ModifyRule* and *RestoreRule* below, all the rest of functions must be applied only one time and prior to activation of the rule processor. If they are applied after activation of the rule processor the result and behavior of the system would be unpredictable if it does not cause crashes. If they are not applied the predefined values would be used.

SetRuleMax(*MaxNoOfRules*): sets the maximum allowed number of rules for the rule repository to the value of *MaxNoOfRules*. *MaxNoOfRules* must be a positive integer value.

SetRepository(*RepositorySize*): sets the maximum allocation size of the rule repository to the value of *RepositorySize*. *RepositorySize* must be a positive integer value.

SetRuleLen(*RuleLength*): sets the maximum rule length to the value of *RuleLength*. RuleLength must be a positive integer.

SetRuleFName(*ApplRules*): notifies the rule processor to load the rules that are stored in the file referenced by *ApplRules*. The rule processor prior to activation of Yashar engine will load the rule repository with the rule definitions provided by *ApplRules*. The default file that will be searched for loading the repository is *InterpCmd.text*.

ModifyRule(*RuleLabel,NewRuleStr*): modifies the current definition of the rule referenced by *RuleLabel* to the new definition referenced by *NewRuleStr*. *RuleLabel* must be a value in the range of 0 to 255 and does not exceed the maximum number of allowed rules in the repository. *NewRuleStr* is a pointer to a character string containing the new rule definition.

RestoreRule(*RuleLabel*): removes the most current modification to the rule referenced by *RuleLabel*. If there has not been any modification the function will do nothing.

B.3 Register Manipulation and Miscellaneous Routines

These routines provide access mechanism to Scratch Pad areas (registers), and the ability to change the default setting of other predefined values.

GetRegister(*RegisterNo*): returns the current value of the register referenced by *RegisterNo*. The return value is four byte long and can be casted to a pointer or a long integer in C. RegisterNo must be a positive integer within the predefined range of 1 to 40.

PutRegister(*RegisterNo*,*RegisterValue*): sets the value of register referenced by *RegisterNo* to the value of *RegisterValue*. RegisterValue can be any thing at the most four bytes long. RegisterNo must be a positive integer within the predefined range of 1 to 40.

SetIndent(*UnitSize*): Set the default indentation unit length to *UnitSize*. UnitSize must be a positive integer value. The new size will be used by @+ and @- in formatting the output text.

Appendix C

Tree Representation Definition For Modula-2

The format used to describe tree representation of Modula-2 source is as follows:

$$tree \rightarrow TREE - EXP$$

where tree is the name of a tree and TREE-EXP is the description of the structure of the tree. A TREE-EXP can be either of the form (NODE X Y Z) or of the form X Y Z. A TREE-EXP of the form (node A B C D) means that the tree has the root "node" and has sons X Y Z. A TREE-EXP of form X Y Z means that the tree is actually a forest composed of trees rooted from X Y Z, respectively. In a tree description, a string of upper case letters stands for a tree node with the name of that string. A string of lower case letters stands for a tree whose structure is described by other definitions. The notations [...] and { ...} have their obvious meanings as they are used in BNF definitions.

The language syntax assumed by the Modula-2 Parser is the same as the definition in [Wir83]. The corresponding EBNF grammar rules (marked with ** and the line numbers refer to the line numbers in the appendix of [Wir83]) of the parsing trees are given right before the tree definitions.

```
**1     ident = letter {letter|digit}
ident  --> ID
```

```
**2     number = integer|real
number --> NUMBER

**3     integer = digit{digit}|octalDigit{octalDigit}("B"|"C")|
**4      digit{hexDigit}"H"
**5     real = digit{digit}"."{digit}[ScaleFactor]
**6     ScaleFactor = "E"["+"|"-"]digit{digit}
**7     hexDigit = digit|"A"|"B"|"C"|"D"|"E"|"F"
**8     digit = octalDigit|"8"|"9"
**9     octalDigit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"

**10 string = "'"{charactor}"'"|""{character}""
string --> STRING

**11    Qualident = ident{"."ident}.
qual-ident --> ident
--> (QUALIDENT ident ident)

**12    ConstantDeclaration = ident "=" ConstExpression.
constant-declaration--> (CONSTDCL ident const-expression)

**13    ConstExpression = SimpleConstExpr[relation SimpleConstExpr].
const-expression --> simple-const-expression
--> (relation simple-const-expression simple-const-expression)

**14    Relation = "=" | "#" | "<>" | "<" | "<=" | ">" | ">=" | IN.
```

```
relation --> LESS
--> NOTEQUAL
--> GREATER
--> EQUAL
--> GREQUAL
--> LEEQUAL
--> IN

**15    SimpleConstExpr = ["+" | "-"] ConstTerm {AddOperator Consterm}.
simple-const-expression --> (add-operator first-const-term const-term)
first--const-term --> simple-const-expression
--> (add-operator unary-const-term const-term)
unary-const-term --> const-term
--> (unary const-term)
unary --> UNARYPLUS
--> UNARYMINUS

**16    Addoperator = "+" | "-" | OR.
add-operator --> PLUS
--> MINUS
--> OR

**17    Consterm = ConstFactor {MulOperator ConstFactor}.
const-term --> (mul-operator first-const-factor const-factor)
first-const-factor --> const-term
--> (mul-operator const-factor const-factor)
```

```
**18   MulOperator = "*" | "/" | DIV | MOD | AND | "&".
mul-operator --> TIMES
--> DIVIDE
--> DIV
--> MOD
--> AND
--> AMPERAND

**19    ConstFactor = qualident | number | string | set |
**20     "(" ConstExpression ")" | NOT ConstFactor.
const-factor --> number
--> string
--> qual-ident
--> qual-ident set
--> set
--> expression
--> (NOT const-factor)

**21    Set = [qualident] "{" [element{","element}] "}".
set --> (SET element-list)
element-list --> (LIST {element})

**22    element = ConstExpression [".."ConstExpression].
element --> const-expression
--> (RETICENCE const-expression const-expression)
**23    Typedeclaration = ident "=" type.
type-declaration --> (TYPEDCL ident type)
```

```
**24    type = SimpleType | ArrayType | RecordType | SetType |
**25    PointerType | ProcedureType.
type --> simple-type
--> array-type
--> record-type
--> set-type
--> pointer-type
--> procedure-type

**26    SimpleType = qualident | enumeration | SubrangeType.
simple-type --> enumeration
--> subrange-type
--> (SIMPLETYPE qual-ident)

**27    enumeration = "(" Identlist ")".
enumeration --> (ENUMERATION ident-list)

**28    IdentList = ident {"," Ident}.
ident-list --> (LIST ident {ident})

**29   SubrangeType = "[" ConstExpression ".." ConstExpression"]".
subrange-type --> (SUBRANGE const-expression const-expression)

**30    ArrayType = ARRAY SimpleType{","SimpleType} OF type.
array-type --> (ARRAY simple-type {simple-type} of-type)
of-type --> (OFTYPE type)
```

```
**31    RecordType = RECORD FieldListSequence END.
record-type --> (RECORD field-list-sequence)

**32   FieldListSequence = Fieldlist {";" Fieldlist}.
field-list-sequence --> (LIST field-list {field-list})

**33    Fieldlist = [IdentList ":" type |
**34                CASE [ident ":"] qualident OF variant {"|" variant}
**35                [ ELSE FieldListSequence ] END ].
field-list --> [(VARDCL ident-list)]
--> [(CASEDCL  case-clause variant-list else-clause)]
case-clause --> (CASECLAUSE [qual-ident] quanlident)
variant-list --> (LIST variant {variant})
else-clause --> (ELSE field-list-sequence)

**36    Variant = CaseLabellist ":" FieldListSequence.
variant --> (VARIANT case-label-list field-list-sequence)

**37    CaseLabelList = CaseLabels {","CaseLabels}.
case-label-list --> case-labels {case-labels})

**38    CaseLabels = ConstExpression[".." ConstExpression].
case-labels --> const-expression
--> (RETICENCE const-expression const-expression)

**39    SetType = SET OF SimpleType.
```

```
set-type --> (SETOF simple-type)

**40    PointerType = POINTER TO type.
pointer-type --> (POINTERTO type)

**41    ProcedureType = PROCEDURE [FormalTypeList].
procedure-type --> (PROCEDURE formal-type-list)

**42    FormalTypeList = "("[[VAR] FormalType {","[VAR] FormalType}]")"
**43                     [":"qualident].
formal-type-list --> (LIST {var-formal-type} [return-type])
var-formal-type --> formal-type
--> (FVAR formal-type)
return-type --> (RETTYPE qual-ident)

**44    VariableDeclaration = Identlist ":" type.
variable-declaration--> (VARDEL iden-list type)

** 45    Designator = qualident {"."ident | "["Explist"]" | "^"}.
designator --> (INDEX first-index exp-list)
--> (CARET first-caret)
--> (PERIOD first-period ident)
first-caret --> qual-ident
--> (CARET first-caret)
first-index --> qual-ident
--> (INDEX first-index exp-list)
first-period --> qual-ident
```

```
--> (PERIOD first-period ident)

** 46   Explist = expression {"," expression}.
exp-list --> expression
--> (LIST expression expression {expression})

** 47   Expression = SimpleExpression [relation SimpleExpression].
expression --> simple-expression
--> (relation simple-expression simple-expression)

** 48   SimpleExpression = ["+"|"-"] term {addoperator term}.
simple-expression --> (add-operator first-term term)
first-term --> simple-expression
--> (add-operator unary-term term)
unary-term --> term
--> (unary term)

** 49   term = factor {MulOperator factor}.
term  --> (mul-operator first-factor factor)
first-factor --> term
--> (mul-operator factor factor)

** 50   factor = number | string | set | designator[ActualParameters]
|
** 51            "("expression ")" | NOT factor.
factor --> number
--> string
```

```
--> qual-ident set
--> set
--> designator
--> function-call
--> expression
--> (NOT factor)

string --> STRING
function-call --> (PROCCALL designator actual-parameters)

** 52   ActualParameters = "(" [Explist] ")".
actual-parameters -->  (ACTALPARA [exp-list])

** 53   statement = [assignment | ProcedureCall | IfStatement |
** 54                CaseStatement | WhileStatement | RepeatStatement|
** 55                LoopStatement | ForStatement | WithStatement|
** 56                EXIT | RETURN[expression] ].
statement --> assignment
--> procedure-call
--> if-statement
--> case-statement
--> while-statement
--> repeat-statement
--> loop-statement
--> for-statement
--> with-statement
--> exit
```

```
--> return-statement

exit --> (EXIT)
return --> (RETURN [expression])

** 57   assignment = designator ":=" expression.
assignment --> (BECOME ident expression)

** 58   ProcedureCall = designator [ActualParameters].
procedure-call   --> (PROCCALL designator [actual-parameters])

** 59   StatementSequence = statement{";"statement}.
statement-sequence --> (LIST {statement})

** 60   IfStatement = IF expression THEN StatementSequence
** 61                 {ELSIF expression THEN StatementSequence }
** 62                 [ELSE StatementSequence] END.
if-statement  --> (IF expression then {elsif} [else])
then --> (THEN statement-sequence)
elsif --> (ELSIF expression statement-sequence)
else --> (ELSE statement-sequence)

** 63   CaseStatement = CASE expression OF case {"|" case}
** 64                   [ELSE StatementSequence] END.
case-statement --> (CASE expression case-list [else-part])
case-list --> (LIST case {case})
else-part --> (CASEELSE statement-sequence)
```

```
** 65   case = CaseLabelList ":" StatementSequence.
case --> (CASEL case-label-list statement-sequence)

** 66   WhileStatement = WHILE expression DO StatementSequence END.
while-statement --> (WHILE expression statement-sequence)

** 67   RepeatStatement = REPEAT StatementSequence UNTIL expression.
repeat-statement --> (REPEAT statement-sequence expression)

** 68   ForStatement = FOR ident ":=" expression TO expression
** 69                  [BY ConstExpression] DO StatementSequence END.
for-statement --> (FOR ident expression expression [by-part] do-part)
by-part --> (BY const-expression)
do-part  --> (DO statement-sequence)

** 70   LoopStatement = LOOP StatementSequence END.
with-statement --> (LOOP statement-sequence)

** 71   WithStatement = WITH designator DO StatementSequence END.
with-statement --> (WITH designator statement-sequence)

** 72   ProcedureDeclaration = ProcedureHeading ";" block ident.
procedure-declaration--> (PROCEDURE procedure-heading block ident)

** 73   ProcedureHeading = PROCEDURE ident[FormalParameters].
procedure-heading  --> ident
```

```
--> ident formal-parameters

** 74   block = {declaration} [BEGIN StatementSequence] END.
block --> (BLOCK dcl body)
del --> (DECLARATION {declaration})
body --> (BODY statement-sequence)

** 75   declaration = CONST {ConstDeclaration";"}     |
** 76                 TYPE  {TypeDeclaration";"}      |
** 76                 VAR   {VariableDeclaration ";"} |
** 77                 ProcedureDeclaration ";"        |
** 78                 ModuleDeclaration ";" .
declaration --> (CONST {constant-declaration})
--> (TYPE {type-declaration})
--> (VAR {variable-declaration})
--> procedure-declaration
--> module-declaration

** 79   FormalParameters = "(" [FPSection {";"FPSection}] ")"
** 80                       [":" qualident] .
formal-parameters --> (FPARAM {FPSECTION} [return-type])
return-type --> (RETTYPE qual-ident)

** 81   FPSection = [VAR] IdentList ":" FormalType.
fp-section --> (FPSECTION [var] {ident} formal-type)
var  --> FPVAR
```

```
** 82   FormalType = [ARRAY OF] qualident.
formal-type --> (ARRAY qual-ident)
--> qual-ident

** 83   ModuleDeclaration = MODULE ident[priority] ";"{import}
** 84                        [export] block ident.
module-declaration --> (MODULEDEF ident priority {import} export block
ident)

** 85   priority = "[" ConstExpression "]".
priority --> (PRIORITY const-expression)

** 86   export = EXPORT [QUALIFIED] IdentList";".
export -->
--> (EXPORT [qualified] {ident})
qualified --> QUALIFIED

87    import = [FROM ident] IMPORT identlist";".
import --> (FROM ident single-import)
--> single-import
single-import --> (IMPORT {ident})


** 88   DefinitionModule = DEFINITION MODULE ident ";" {import}
** 89                      [export] {definition} END ident"." .
definition-module --> (DEFINITION ident {import} export {definition} ident)
```

```
** 90   definition = CONST {ConstantDeclaration ";"}|
** 91                TYPE  {ident ["="type ]";"}     |
** 92                VAR   {VariableDeclaration";"} |
** 93                ProcedureHeading";".
definition --> (CONST {constant-declaration})
--> (TYPE {ident} type)
--> (VAR {variable-declaration})
--> procedure-heading

** 94   ProgramModule = MODULE ident [priority] ";" {import}
** 95                    block ident "." .
program-module --> (MODULE ident priority import block ident)

** 96   CompilationUnit = DefinitionModule |
** 97                      [IMPLEMENTATION]  ProgramModule.
tree  --> (ROOT {compilation-unit})
compilation-unit --> definition-module
--> implementation-module
--> program-module
```

C.1 References

[Wir83] Niklaus Wirth. *Programming In Modula-2. Texts and Monographs in Computer Science*, Springer-Verlag, Berlin Heidelberg, 1983.

Appendix D

Generalizer and Refiner Functions

Reusability functions are divided into two groups: 1) functions for Generalization, and 2) functions for Refinement. There is a generalization and refinement support function defined for each language fragment. These functions are installed in the Function Table by the Arash User Interface Routines when Arash is started. The reference index to each function is shown in front of each function name.

The context of the rule processor, a pointer to the rule definition, and a pointer to the tree node that will be generalized or refined are passed to the function when activated by the Rule Processor. These functions are assumed to return a pointer to a character string as the result of their activation. If nothing is to be returned, a null pointer is returned.

A user can alter the semantics of each of the generalization and refinement functions by installing his own.

D.1 Generalization support Functions

As part of their activities each of these functions produce the meta identifiers for the constructs that they support.

0 **MtConsG()**: Generalization function for constant fragments.

1 MtTypeG(): Generalization function for type fragments.

2 MtVarG(): Generalization function for variable declaration fragments.

3 MtPrcG(): Generalization function for procedure declaration fragments.

4 MtAssG(): Generalization function for assignment statement fragments.

5 MtPCallG(): Generalization function for procedure call fragments.

6 MtIfG(): Generalization function for if fragments.

7 MtCaseG(): Generalization function for case fragments.

8 MtWhileG(): Generalization function for while fragments.

9 MtRepeatG(): Generalization function for repeat call fragments.

10 MtForG(): Generalization function for for fragments.

11 MtLoopG(): Generalization function for loop fragments.

12 MtWithG(): Generalization function for with fragments.

13 MtReturnG(): Generalization function for return fragments.

D.2 Refinement Support Functions

The refinement support functions operate on the abstracted fragments to create a concrete instance. If the .MLS file exists for the abstracted module under refinement its functionality is extended to perform extra steps as explained in Refiner Operation section.

0 CuConsG(): Refinement function for constant fragments.

1 CuTypeG(): Refinement function for type fragments.

2 CuVarG(): Refinement function for variable declaration fragments.

3 CuPrcG(): Refinement function for procedure declaration fragments.

4 CuAssG(): Refinement function for assignment statement fragments.

5 CuPCallG(): Refinement function for procedure call fragments.

6 CuIfG(): Refinement function for if fragments.

7 CuCaseG(): Refinement function for case fragments.

8 CuWhileG(): Refinement function for while fragments.

9 CuRepeatG(): Refinement function for repeat call fragments.

10 CuForG(): Refinement function for for fragments.

11 CuLoopG(): Refinement function for loop fragments.

12 CuWithG(): Refinement function for with fragments.

13 CuReturnG(): Refinement function for return fragments.

Appendix E

Rules To Recognize Modula-2 Source

These rules are needed to reproduce the original Modula-2 source program text from the Tree Representation in main memory. In some cases, no rule is needed, in which case the null rule **NA** is used.

```
0:
1:ARRAY @D/,/@01 OF @02
2:***NA***2
3:BY @01
4:CASE @01 OF @n@+@D/ |@n/@02@n@+@03@-@-@nEND
5:CONST @D/;@n/@n@+@*//;@n@-
6:DEFINITION MODULE @01;@n@*//.@n
7:@01 DIV @02
8:DO @n@+@D/;@n/@01
9:@-ELSE@n@+@D/;@n/@01@-@+
10:@-ELSIF @01 THEN@n@+@D/;@n/@02@*/;@n/@-@+
11:END
12:EXIT
13:EXPORT @D/,/@01;
14:FOR @01 := @02 TO @03 @04 @05 @-@nEND
```

```
15:@+FROM @O1@O2@n@-
16:IF @O1 THEN@+@*/@n/@n@-END
17:IMPLEMENTATION MODULE @O1@O2@*//.@n
18:@+IMPORT @D/,/@O1;@n@-
19:@O1 IN @O2
20:LOOP@n@+@D/;@n/@O1@-@nEND
21:@O1 MOD @O2
22:MODULE @*//
23:NOT @O1
24:***NA***24
25:@O1 OR @O2
26:***NA***26
27:PROCEDURE @O1@*//
28:QUALIFIED @D/,/@O1;
29:@n@+RECORD @n@+@D/;@n/@*//@n@- END@-
30:REPEAT @n@+@D/;@n/@O1@n@-UNTIL @O2
31:RETURN @*//
32:{@D/,/@O1}
33:@D/;@n/@O1
34:***NA***34
35:TYPE@D/;@n/@n@+@O1;@n@-
36:***NA***36
37:VAR @D/;@n/@n@+@*//;@n@-
38:WHILE @C+@O1@D/;@n/@C- DO @+@n@O2@n@-END
39:WITH @O1 DO @n@+@D/;@n/@O2@-@nEND
40:@O1
41:"@d"
```

```
42:@d
43:@d
44:;@n
45:***NA***45
46:***NA***46
47:@01.@02
48:@01..@02
49:@Cm(@01 < @02)@Cm
50:(@01 > @02)@Cm
51:(@01 = @02)@Cm
52:(@01 >= @02)@Cm
53:(@01 # @02)@Cm
54:(@01 <= @02)@Cm
55:@^1@(@01 + @Cm@02@)
56:@^2@(@01 / @Cm@^3@02@^2@)
57:@^2@(@01 * @Cm@02@)
58:@^1@(@01 - @Cm@^2@02@^1@)
59:@Cm@01 & @02@Cm
60:***NA*60
61:***NA*61
62:***NA*62
63:***NA*63
64:***NA*64
65:***NA*65
66:@01 := @02
67:***NA*67
68:@01^
```

```
69:***NA*69
70:@01@n@02
71:@*/@n/
72:(@D/,/@01)
73:@D/./@01
74:ARRAY@D/,/@01 OF @02
75:@01@*//
76:(@D/; /@01)@02;@n
77:@D/,/@01:@02
78: [@01];@n
79:@01@*D
80:@01[@02]
81:@^O@(+@01@)
82:@^O@(-@01@)
83:@01 : @n@+@D/;@n/@02@-
84:BEGIN@n@+ @D/;@n/@01@*D@-@nEND
85:@01
86:@01 = @02
87:(@D/,/@01)
88:@D/,/@01 : @02
89:SET OF @01
90:POINTER TO @01
91:@D/,/@01 : @n@+@D/;@n/@02@-
92:[@01..@02]
93:***NA*93
94:@D//:@01
95:@D/./@01
```

```
96:@01 : @02
97:@01@*//
98:CASE @01 OF @n@+@D/ |@n/@02@n @03 @-END
99:@01 = @02
100:VAR @D/,/@01@02
101:PROCEDURE @D/, /(@01)@02
102:VAR @01
103:@D//:@01
104:@-ELSE@n@+@D/;@n/@01@-@+
```

Appendix F

Generalizer Selection Dialogs

This section contains dialog boxes used to select language fragments to be Refined.

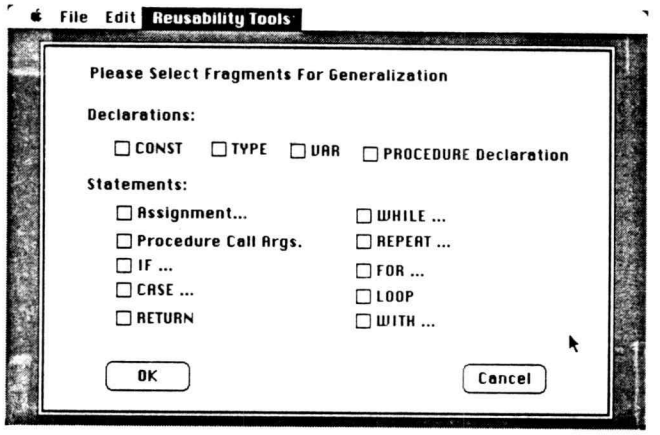


Figure F.1: Selection Dialog for Modula-2 Language Fragments

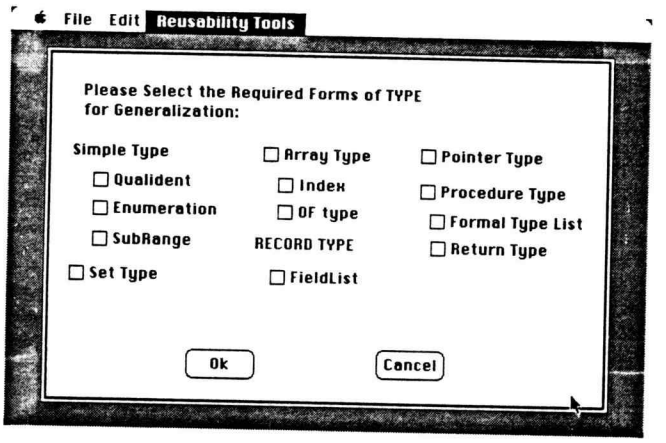


Figure F.2: Selection Dialog for Type Fragments

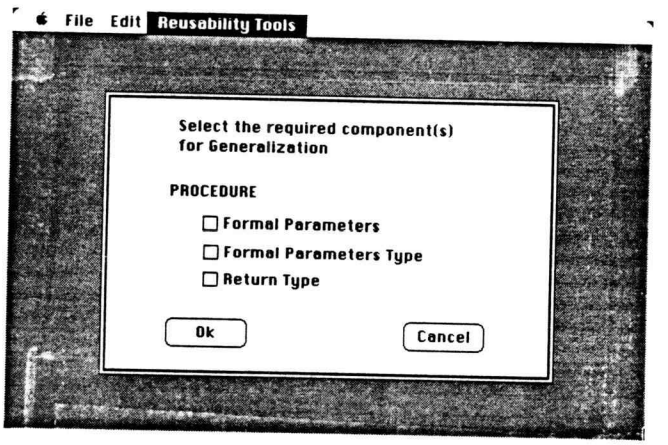


Figure F.3: Selection Dialog for Procedure Declaration Fragment

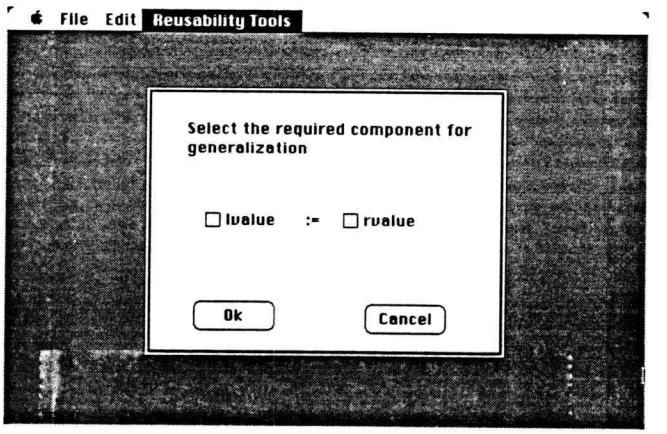


Figure F.4: Selection Dialog for Assignment Fragment

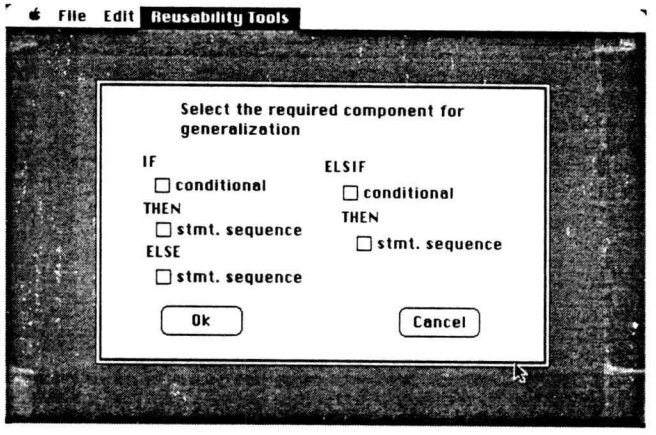


Figure F.5: Selection Dialog for IF Fragment

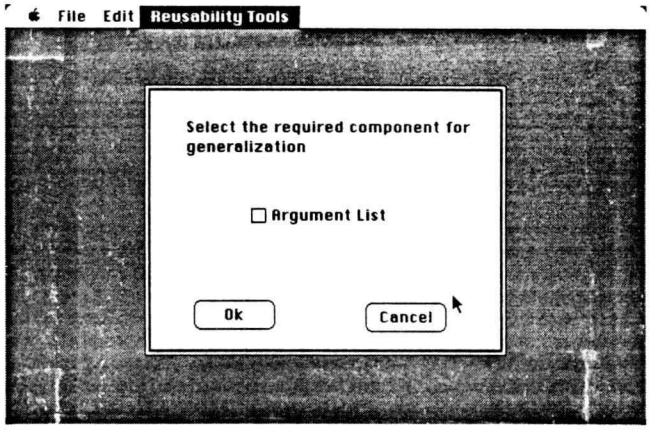


Figure F.6: Selection Dialog for Procedure Call Fragment

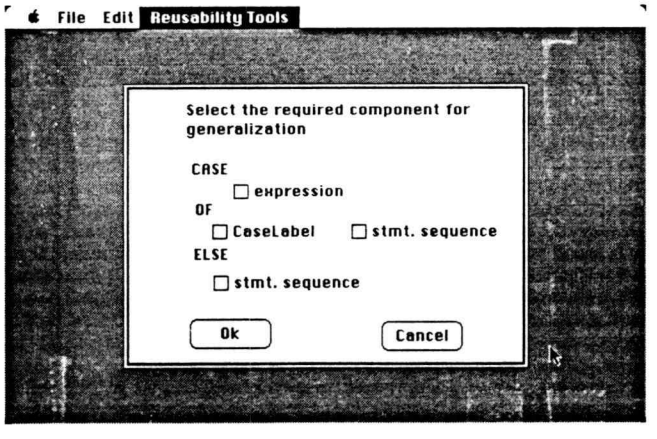


Figure F.7: Selection Dialog for CASE Fragment

Figure F.8: Selection Dialog for WHILE Fragment

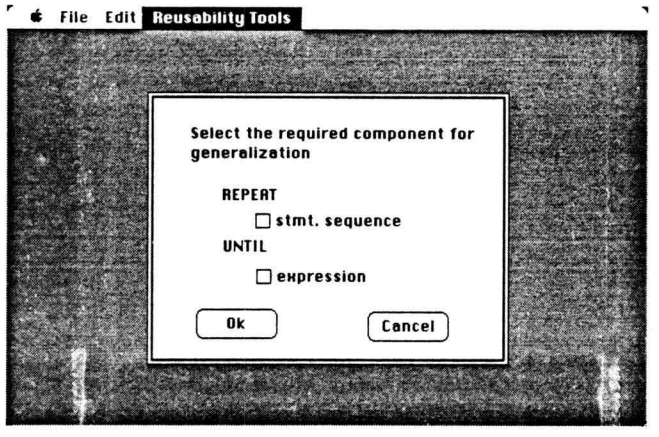


Figure F.9: Selection Dialog for REPEAT Fragment

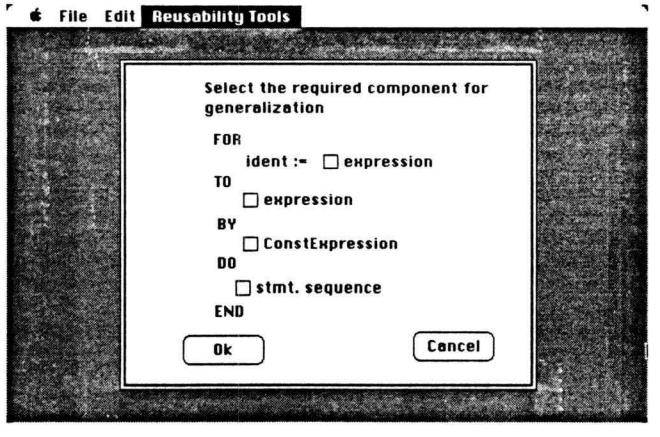


Figure F.10: Selection Dialog for FOR Fragment

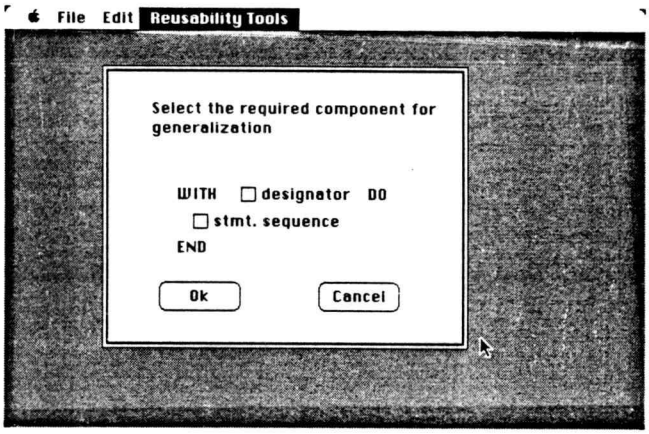


Figure F.11: Selection Dialog for WITH Fragment