

OREGON STATE

UNIVERSITY

COMPUTER

SCIENCE

DEPARTMENT

YASHAR: A RULED BASED META-TOOL FOR
PROGRAM DEVELOPMENT

Abbas Birjandi

T. G. Lewis

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

(503) 754-3273

TR-86-30-⁶~~3~~

10-1

Yashar: A Ruled Based Meta-Tool For Program Development

Abbas Birjandi

T.G. Lewis

Department of Computer Science

Oregon State University

Corvallis, Oregon 97331

(503) 754-3273

,1.11.5

Abstract:

Yashar is a rule based meta-tool for rapidly producing tools to increase programming speed through automating restructuring of existing source code modules so they can be reused, generating of syntax-directed tools, language-to-language translation, automated document generation , and various debugging tools. The main significance of Yashar is that it can be tailored to a wide variety of applications through the specification of rules. In this paper we describe the rule processor, rule syntax, and how to create an instance of Yashar for translating Modula-2 source programs into equivalent C source programs, automatically. Finally, we conclude that 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 application-specific interface and support routines.

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

1 Introduction

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

Two distinct approaches have been taken in building programming environments: 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]. The Unix operating system contains many examples of tool boxes. For example, the *MAKE* utility [Fel79] is a useful tool for managing the compile-link phase of implementation. A programmer must know how to set-up, apply, and understand the output produced by *MAKE*.

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

Yashar is a meta-tool for generating programming environment tools. A tool produced by Yashar takes advantage of both approaches, but it 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 tree-structured representation of its input. However, unlike interlisp, a

Yashar tool is not bound to one specific language and can support multiple languages within the same programming paradigm. Thus, Yashar is 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 tool is an *instance* of the meta-tool. It is our intention to show how *instantiation* of a meta-tool such as Yashar can be of benefit to both tool developers and software developers, alike.

The main significance of Yashar is that it is a meta-tool for generating an application-specific tool through specification of rules. These rules are interpreted by a generalized rule processor which transforms the tree-structured inputs into useful outputs. It is because of these rules that we call an instance of Yashar a rule-based environment. The usefulness of this approach is the central theme of this paper.

Our notion of a rule differs from the notion used in logic programming [CM84]. In logic programming, a rule states a proposition corresponding to a logical implication [Col85]. In other words rules can be viewed as a formalism for defining knowledge independent of the method of computation. In contrast, Yashar rules are imperative commands which describe the computation itself rather than declarative commands which state some logical implication.

This project is mainly concerned with block structured programming languages and syntax directed processing of modular languages such as Ada[DOD82], Pascal[JW76], and Modula-2[Wir83]. Yashar is written in C and runs on the Apple Macintosh personal computer—thus converting a Macintosh into a low-cost programmer's workstation.

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. A practical meta-tool is one that can be tailored into many useful tools. A tool is an instance of Yashar, where an instance is obtained by binding Yashar's rules and rule-processor to a certain application-specific domain such as suggested by the list of tools below:

- A tool for building reusable software components
- A language directed editor
- An incremental code generator
- A language-to-language translator
- A source code debugger and performance monitor
- A structured document generator
- An adaptive language based prettyprinter

Instances of Yashar which we have studied are: 1) building reusable source programs by incorporating application interface routines called a **Generalizer** and a **Refiner**, 2) creation of an adaptive prettyprinter for Modula-2, 3) creation of a structured document processor, and 4) creation of a language-to-language translator. In addition it is our belief that Yashar is a viable meta-tool for building language directed editors, incremental code generators, and other syntax based tools. These will be reporting on in a later paper.

2 Overall System Architecture

There are two class 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 C routines to perform the following functions shown in Figure 1:

- Application Interface Routines
- Internal Data Model Builder

In addition, a designer must write a series of rules which are stored in a *rule repository*, see Figure 1. These rules are actions to be carried out by the Yashar Rule Processor when a programmer uses Yashar as a tool. There are two kinds of rules stored in the rule repository: 1) application-specific rules, and 2) user-defined rules. All application-specific rules are written by the designer and installed in the rule repository by hand when instantiating Yashar. All user-defined rules are generated automatically by Application Interface Routines, which are written by the designer. That is, user-defined rules are written by user-defined routines which are installed by a designer. Figure 1 shows the parts of Yashar which must be installed manually

Figure 1: A Generic Yashar Programming Environment

by someone who creates a tool as an instance of Yashar.

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 Internal Data Model Builder

into a Yashar tree structure, and if appropriate, into Data Dictionary information. The programmer controls this process through the user's interface specified by the Application Interface Routines. Recall that these routines are specific to an instance of Yashar, and they may change from one instance to another. A collection of Application Interface Routines for a Modula-2 to C translator tool, for example, will differ from the Application Interface Routines for a tool that restructures reusable modules.

Next, a programmer directs the tool to process the tree structured input file by requesting actions from the Rule Processor. In Figure 1, the Rule Processor obeys commands from the programmer, takes rules from the Rule Repository, and applies these rules to the tree representation of the input. In some application, a tool might require additional information from the Data Dictionary, also shown in Figure 1.

Figure 2: A Modula-2 to C Translator Tool

An instance of Yashar is shown in Figure 2, which contains application-specific versions of the components shown in Figure 1. In Figure 2 a designer has written Application Interface Routines to perform the Modula-2 to C Translator User Interface functions; written an Internal Data Model Builder called the Modula-2 Internal Data Model Builder to parse Modula-2 source code and store it as an internal tree; and finally, supplied both application-specific and user-defined rules for recognizing Modula-2 statements and then re-writing them as equivalent C statements.

The input to Figure 2 is a Modula-2 source program file and the output is an equivalent C source program file. Input and output processing is done by the Rule Processor, which takes commands from the Rule Repository, operates on the tree representation of the Modula-2 sources code, accesses symbol table information stored in the Data Dictionary for Modula-2 identifiers, and returns the results to

the Modula-2 to C Translator User Interface.

3 Internal Structure of Yashar

In order to understand how Yashar works, a designer must understand three central structures:

- The Internal Tree and Data Dictionary containing input document data,
- The Rule Processor , Rule Repository, and Syntactic and Semantic structure of Rules,
- The Interaction between Rules, Rule Processor, and the Internal Tree/Data Dictionary.

These three structures, and their associated processing routines, must be set-up by a tool designer. This is a *one-time* only step, and once accomplished, it is not necessary for a programmer to be familiar with them.

3.1 Internal Tree Structure

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 an instance of Yashar. Trees were chosen as Yashar's data model because they are capable of expressing complex information and at the same time are simple enough to be handled algorithmically by a computer. For example, source programs can be easily converted to a tree representation before being manipulated by Yashar's rule processor.

All inputs to Yashar are first converted to a tree structure by the Internal Data Model Builder as shown in Figure 1. The Internal Data Model Builder is written in C by the tool designer and combined with other C routines to be compiled into an instance of Yashar. For example, Figure 3 shows a Modula-2 source code program as it is converted into a tree structure by the Internal Data Model Builder.

Figure 3: Internal representation of an object in Yashar

Each node of the tree holds four categories of information:

Label Info.: Each node is assigned a label which indicates its type. Type Information is used by the rule processor to decide what rule to apply to the values stored at each node. In Figure 3 node types 17, 43, 70, ... refer to different logical parts of a typical Modula-2 source program. Note that the tree representation of the sample has some extra node types (45, 43 second subtree of node 88) not usually found in an abstract syntax tree. The extra nodes merely make translation to C and regeneration of Modula-2 easier.

Data Dictionary Reference Pointer: The data dictionary captures and disseminates information related to attributes of the nodes of the tree. For example, the Modula-2 to C translator in Figure 2 uses a data dictionary to hold the type, scope, and value of variables declared in the original source program. The scope of the main module is assumed to start from 1, so the scope value for *W* will be 1 in the data dictionary.

Link Information: Link information maintains the internal representation of a tree. Children of each node are numbered sequentially from the left to right. This *sequence number* is used to access a child of a node. For example, in Figure 3 node label 17 has four children that are numbered from left to right

as 1 (node 43), 2 (node 44), 3 (node 70), and 4 (node43). Internally, general n-ary trees like the one shown in Figure 3 are maintained as a binary tree based on the natural transformation in [Knu73].

Application data/linkage: The exact amount of detail stored in the tree depends on the nature of the specific application. In Figure 3 this field was not used, but in some other example this pointer would reference additional information. For example, the tree might represent an abstract syntax tree which has been augmented with extra information needed to regenerate the original source program from the tree. Or, the tree might be used in a language directed editor to maintain a mapping between a node of the program tree and the X-,Y-Coordinates of the text on the screen. Thus, each node of the tree contains an application data/linkage field that is used to either hold user-defined values or extend the data structure of each node without altering other portions of the system. All such application specific data can be captured in this user defined data structure, and associated with the corresponding tree nodes.

3.2 Internal Data Model Builder

The Internal Data Model Builder produces a tree representation of the input text. A special purpose Internal Data Model Builder must be hand-crafted for every instance of Yashar. To facilitate this work, there is a set of pre-compiled 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 through the use of Application data/linkage field and/or the application defined data structure.

In general there is no specific requirement in building the tree. A useful guideline

in deciding the logical order of tree nodes and the amount of information stored at each node is to consider what is needed to write rules which regenerate the input. Caution should be exercised, however, because keeping extra information to make writing the rules easier may result in creating unnecessary node types and adding extra storage and processing time overhead. The second useful guideline is to consider the effort needed to automatically generate rules in an interactive environment.

3.3 Yashar's Rule Processor

Yashar's Rule Processor is a transformational unit which converts input stored in the tree representation into various kinds of output. Figure 4 is the execution environment of Yashar's rule processor. Rules direct Yashar's rule processor to perform transformational operations on an input tree. The transformational operations are carried out by either application-specific rules or by user-defined rules. The application-specific rules are installed in the Rule Repository by a designer prior to instantiating Yashar. The user-defined rules are installed in the Rule Repository *on the fly* by user-defined support functions. A user-defined support function is a C

Figure 4: Yashar's Rule Processor Execution Environment

routine written and installed by the designer of an instance of Yashar. Such a function is executed as a consequence of a reference to its *function table number*. This number is also assigned to the user-defined support routine by a designer. When called, a user-defined support function may modify an existing application-specific rule, or write an entirely new rule.

Function Table

Function Table is a vector that stores the address of different user-defined support functions for execution. All the user-defined support functions must be installed in the Function Table prior to activation of the rule processor.

Rule Repository

The *Rule Repository* is the storage element where application-specific and user defined rules are stored and accessed by the rule processor. This storage element is

divided into two logical parts. The first part contains application-specific rules and the second part stores user-defined rules.

Application-Specific Rules

Application-specific rules define the default sequence of processing and semantic actions of the rule processor when processing the tree-structured input. For example, in an adaptive prettyprinter for Modula-2, the application-specific rules govern the traversal of the tree and tell how to produce the formatted output text.

User-Defined Rules

User-defined rules modify or augment existing application-specific 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 an application-specific rule can be reversed. The rule processor keeps a copy of the original rule and simply removes the most recent modification.

Rule Processing

The input to Yashar's rule processor is a tree-representation of the application-specific input, application-specific rules, and possibly a set of user-defined rules.

Scratch Pad Area

The Scratch Pad Area is a set of *Registers* used to communicate among user-defined support functions, between user-defined support functions and the rule processor,

and among the rules themselves. Access from within a rule is symbolically referenced as Rn where n is a two digit register number. Currently there are 40 registers ($R01 - R40$) defined for such purposes. Access is also possible from a user-defined function through two pre-compiled Yashar support routines *GetRegister* and *PutRegister*.

Yashar Pre-compiled and User-defined Support Functions

The *pre-compiled* support functions are used by an instance 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 user-defined support functions and Application Interface Routines. Refer to Appendix B for a list of *pre-compiled* support functions.

The *user-defined* support 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 C 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.

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. 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.

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@04

The rule processor reads this rule and does the following:

- Process the first child of node 39 (@01)
- Do not process the second child of node 39 (@I02)
- Process the third child of node 39 (@03)
- Process the fourth child of node 39 (@04)

The above sequence of activities applies to all nodes with label 39. 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. See the explanation of @ⁿ in Appendix A.

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. See the explanation of @*D* in Appendix A.

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.

3.3.1 Rule Processor Instruction Set

For a complete list of the instruction set of Yashar's rule processor and their semantic definition refer to Appendix A. The instruction set of Yashar's 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, pre-compiled Yashar support functions, and user-defined 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:

$$\begin{array}{ccccccc}
 \textit{label} & & \textit{FormattingInst.} & & \textit{FormattingInst.} & & \\
 \underbrace{02} & : & \underbrace{BEGIN} & \underbrace{@n@+} & \underbrace{@M\$R03 = (@01)\$@01} & \underbrace{@-} & \underbrace{END} \\
 & & \textit{text} & & \textit{activeInst.} & & \textit{text}
 \end{array}$$

directs the rule processor to transform every node of type 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 node of type 02.
- Decrement the indentation level by one increment unit (@-)
- Emit an **END**

It is important to note the difference between rules in Yashar and syntax definitions that are used in syntax directed editors. 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 statements of its body.

added for hiding details

38 : *WHILE*@C+@01@D/;@n/@C-DO@+@n@An/ < whilebody > /@02 @n@- *END*

Assuming that there is a while loop with ten statements in its body the following is what would be seen after the above rule takes effect.

Before	After
Detailed While loop	While loop abstraction without details of its body
<pre> WHILE a <= b DO a := a +1; IF b <> 0 THEN ... : END </pre>	<pre> WHILE a <= b DO <whilebody> 10 END </pre>

The following is an explanation of the rule:

- Emit a **WHILE** (*WHILE*)
- Activate conditional filling (@C+)
- Process the first child if 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)

Tree Navigational Instructions

The order in which children of a tree node are processed is specified by a sequence number prefixed with an @. For example the following causes the rule processor to process first, second, and forth children of all nodes labeled 20 respectively.

20 : @01@02@/03@04

Formatting Instructions

Formatting instructions are for prettyprinting of 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.

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

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 user-defined support function to be executed. For example the following directs the rule processor to skip the second child of every node whose label equals 34 and to pass control and context of the rule processor to the 5th function in the Function Table.

34 : @01@02%05

The $@Jn$ designates a user-defined support function call instruction. The rule processor executes the n th 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), 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:

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

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:

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

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.

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$IF@01%12THEN@ + @D/;@n/@ * /@n/@n@ - END$...
```

will cause the rule processor to first modify the rule definition 16 to what is enclosed between two \$ delimiters, and then continue with the remainder of rule 30. To restore the original definition of rule 16, some rule must contain the following ...@r16. Alternatively, the *RestoreRule* pre-compiled Yashar support routine can be called from within a user-defined interface or support routine to change rule 16 back to its original form.

3.4 Application Interface Routines

Application Interface Routines are written by the designer and linked to Yashar support routines during the creation stage of an instance of the Yashar. They are generally responsible for:

- Communicating with the user of the environment,
- Generating new application-specific rules automatically,
- Loading of the newly generated rules,
- Activating the rule processor, and
- Capturing the results of the operation

As part of initialization of an instance of Yashar, Application Interface Routines must install the user-defined support functions in the Function Table that are in turn activated by the rule processor automatically when referenced within the rules.

Loading of the newly generated rules could be done either prior to activation of the rule processor through installation of a user-defined support function in a predefined location of Function Table, or interactively by calling the pre-compiled Yashar support routine *ModifyRule* by user-defined application interface routines, or by the rule themselves, or any combination of the above methods.

Changing the default storage size of Rule Repository, rule size, and maximum allowed number of rules in the Rule Repository could be done by calling *SetRepository*, *SetRuleLen* and *SetRuleMax* respectively.

4 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 5 involve the C compiler and linker. This is a *one time only* process which we call *instantiation of a tool*.

Figure 5: Creation of an Application Specific Version of Yashar

First, the tool designer must write application interface routines. This usually involves writing C functions to handle windows, dialogues, and menus. Next, the designer must write application-specific rules which are used by the Rule Processor.

For example, to instantiate Yashar as a Modula-2 to C translator, a designer would have to write interface routines for the keyboard, screen, and mouse, and string processing routines. Figure 2 displays the dataflow and control flow of an instance of Yashar for translating Modula-2 source code programs into semantically equivalent C source code programs.

The Modula-2 to C Translator User Interface is a C program written by a tool designer which provides the communication channel between the user and the translator environment. The Translator User Interface takes care of selecting the Modula-2 source code to be translated, activation of the Modula-2 Internal Data Model Builder, and installation of user-defined support functions prior to activation of rule processor, by requests to translate.

Four user-defined support routines were written; 1) the first function maps the scalar types of Modula-2 that are not supported in C; 2) the second function maps variable parameters to semantically correct C form; 3) the third function to resolve

the scope problem of variables referenced in nested procedures in Modula-2 when they are de-nested for C; and 4) the fourth function accesses the data dictionary to get the variable names and their attributes.

Figure 6 is a sample Modula-2 program and its C program equivalent produced by this instance of Yashar.

Figure 6: A Sample Modula-2 program and its C equivalent

5 Conclusions

Yashar originally started as an experiment in building a tool for program transformation [BGW76,Che83] and understanding 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]. Some of the ideas for prettyprinting source programs are adopted from [Fri83]. The instruction set of Yashar's rule processor provides a powerful mechanism for manipulating structured data and to experiment and build interface tools specially interactive that are mostly useful in program development environments. The terse notation of Yashar's instruction set facilitates automatic generation of the rules by interface programs. The capability of modifying rules during execution is valuable in adjusting the actions of an instance of Yashar to respond to various possible cases, and adds to its versatility. Furthermore, escaping the ordinary sequence of processing of the nodes through escape and break point instructions (`@Jn`, `@n%m`, ...) allows extensibility and enhancement of the functionality of the rule processor. Also breaking instructions provide a simple means of creating interactive environments.

Manual creation of the Application User Interface Routines, Internal Data Model Builder, and writing of user-defined support routines may seem as a drawback , however this is nominal and worth the effort because an entire family of tools are obtained as a result.

A Yashar's Rule Processor Instruction List

This appendix lists the instructions of Yashar's rule processor and their semantics. 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 $\alpha \dots \alpha$ means a string of characters enclosed by two identical symbols. The symbol α can be any printable character. The string must not contain the symbol α . The notation of the instruction is partly influenced by [Fri83]. The meta symbols < and > enclose non-terminals such as arithmetic expressions.

A.1 Formatting Instructions

The @Cm, @C+, and @C- instructions provide a means of adjusting textual output to line size, and number of pixels of the display device. They provide the knowledge to make the best judgement as to where to break the output text.

@+ 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 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 best logical 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.2 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.4. 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{A₁, A₂}** Make a copy of the subtree pointed by **A₁** and set **A₂** to the address of created subtree. **A₁** and **A₂** must be registers only. Not implemented yet.
- @.PT{A₁, A₂, [S|K]n}** Paste subtree pointed by **A₂**, to subtree pointed by **A₁** as **n**th 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.
- @Aα...α@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.
- @ARnα...α@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.
- @Anα...α@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.
- @AnRnα...α@nn** Do as above but also save the number of children of children number **nn** of current node in register number **n**.
- @Dα...α** 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.

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

A.3 Escape and Break point Instructions

@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 n th function in Function Table. No *context* information is passed to the function.

@P $\alpha n \alpha m$ Defines a break point for node type n and designates the m th function in the Function Table to be the breaking function.

@V $\alpha n \alpha$ 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.4 Arithmetic and Conditional Instructions

@?($\langle \text{exp} \rangle$)?^{*true part*} $[n : \text{newrule}]?$ _{*false part*} $[n : \text{newrule}]?$ Conditionally modifies a command depending on whether the condition being test is true or false. The $\langle \text{exp} \rangle$ 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.5 Miscellaneous Instructions

Rule Repository Manipulation Instructions

@Mn α ... 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 user-defined support 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 user-defined support 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 user-defined support routine to modify or augment the application-specific 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, to actually modify the rules.

B Pre-Compiled Support Routines

This section explains the pre-compiled support routines that are available for application-specific interface and user-defined support routine to access the tree-representation

of input, the Scratch Pad area and rule repository.

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*, *Type Value*): set the type of the node pointed by *treenode* to the value of *Type Value*. Currently the *Type Value* 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.

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.txt*.

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.

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.

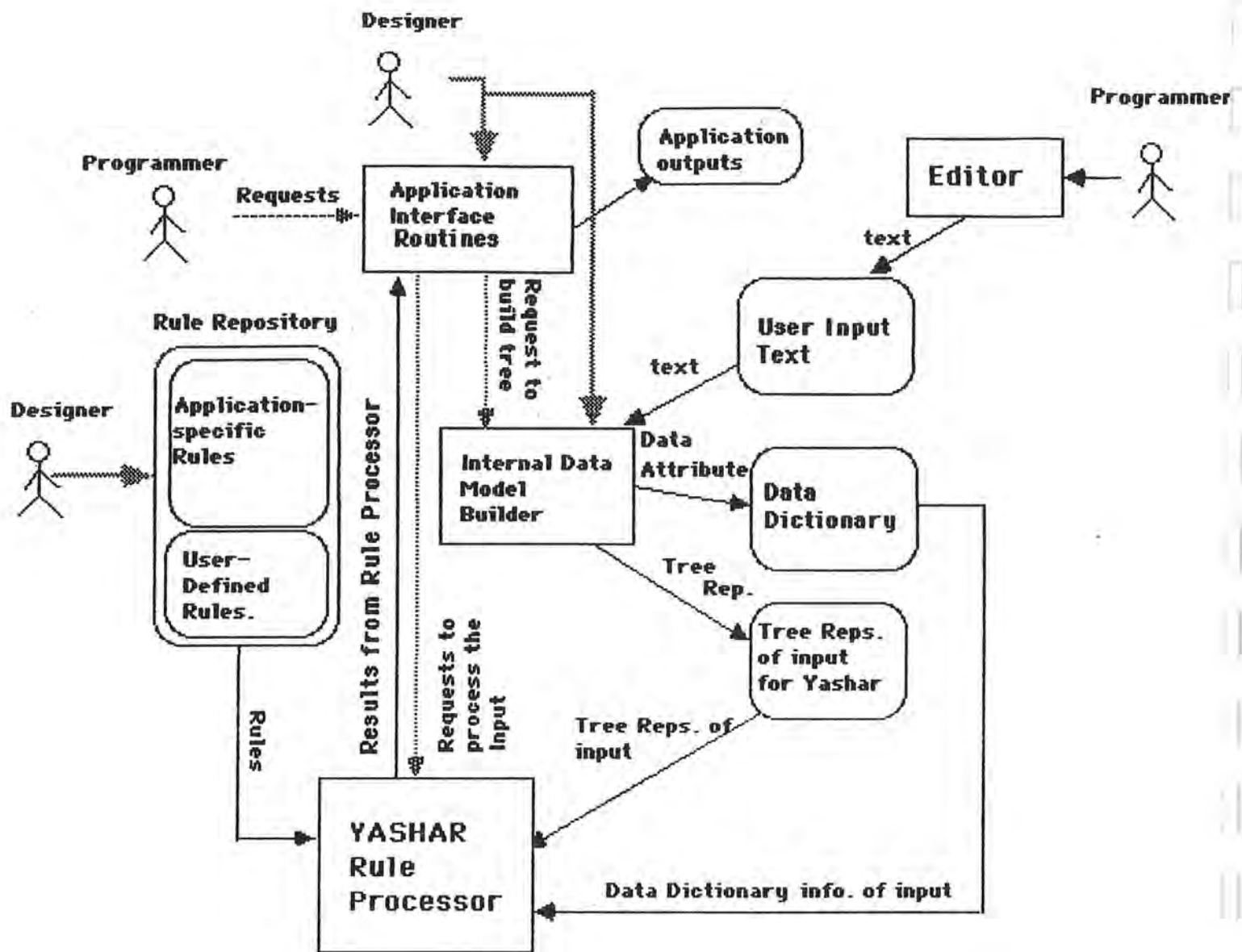
SetIndent(*UnitSize*): Set the default indentation unit length to *UnitSize*. *UnitSize* must be a positive integer value. The new size will be used by @+ and @- in formatting the output text.

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.
- [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.
- [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.
- [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.
- [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.
- [JW76] K. Jensen and Niklaus Wirth. *Pascal User Manual and Report*. *Texts and Monographs in Computer Science*, Springer-Verlag, 2 edition, 1976.
- [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.
- [Ost81] Leon Osterweil. Software environment research: directions for the next five years. *IEEE Computer Magazine*, 14(4):35-43, 1981.
- [TM81] Warren Teitelman and Larry Masinter. The interlisp programming environment. *IEEE Computer Magazine*, 14(4):25-33, 1981.

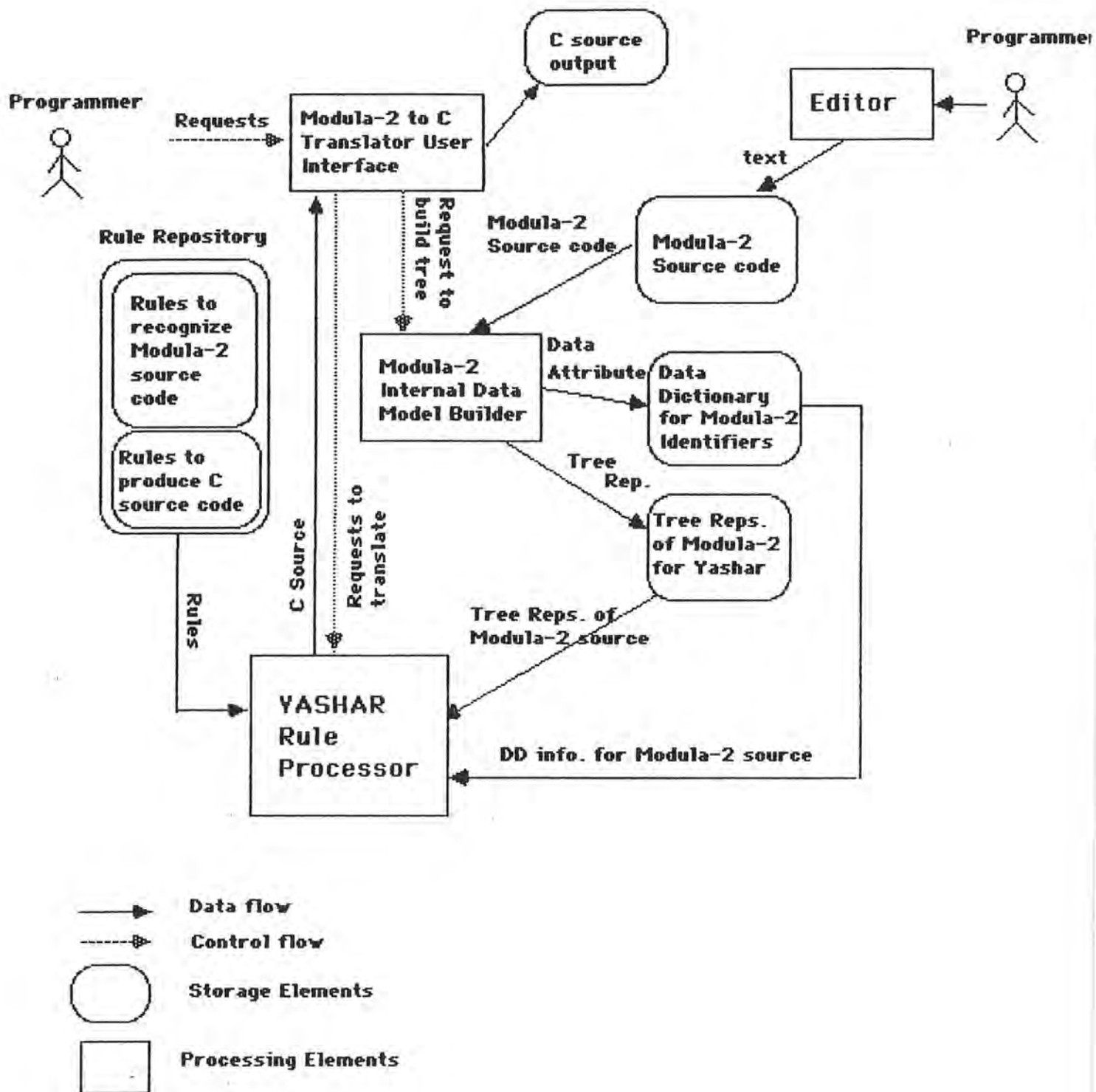
- [TR80] Tim Teitelbaum and Thomas Reps. *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*. Technical Report, Computer Science Dept. Cornell University, MAY 1980.
- [Wir83] Niklaus Wirth. *Programming In Modula-2. Texts and Monographs in Computer Science*, Springer-Verlag, Berlin Heidelberg, 1983.



-  installation
-  Data flow
-  Control flow

 Storage Elements

 Processing Elements



A Modula-2 to C Translator

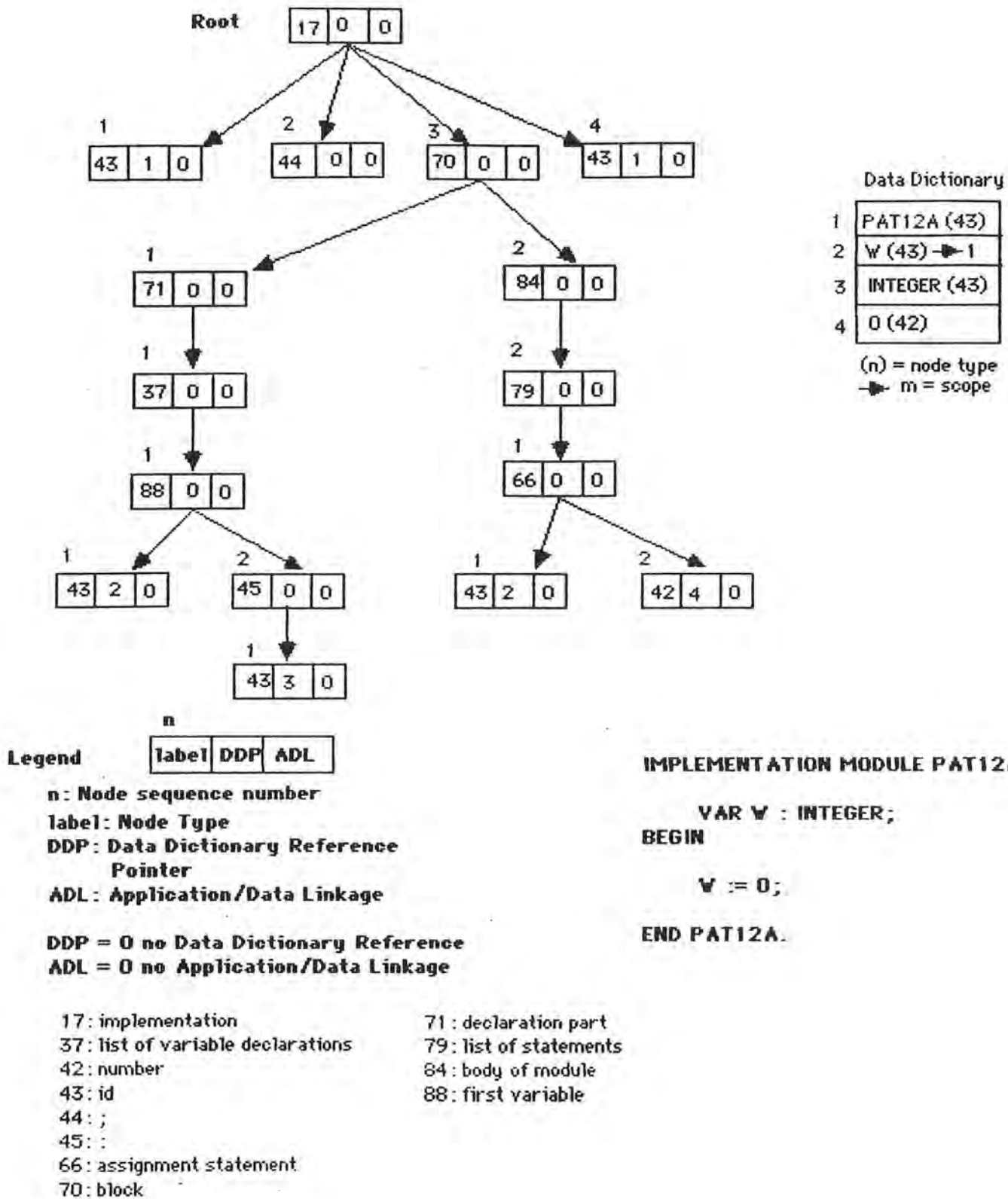
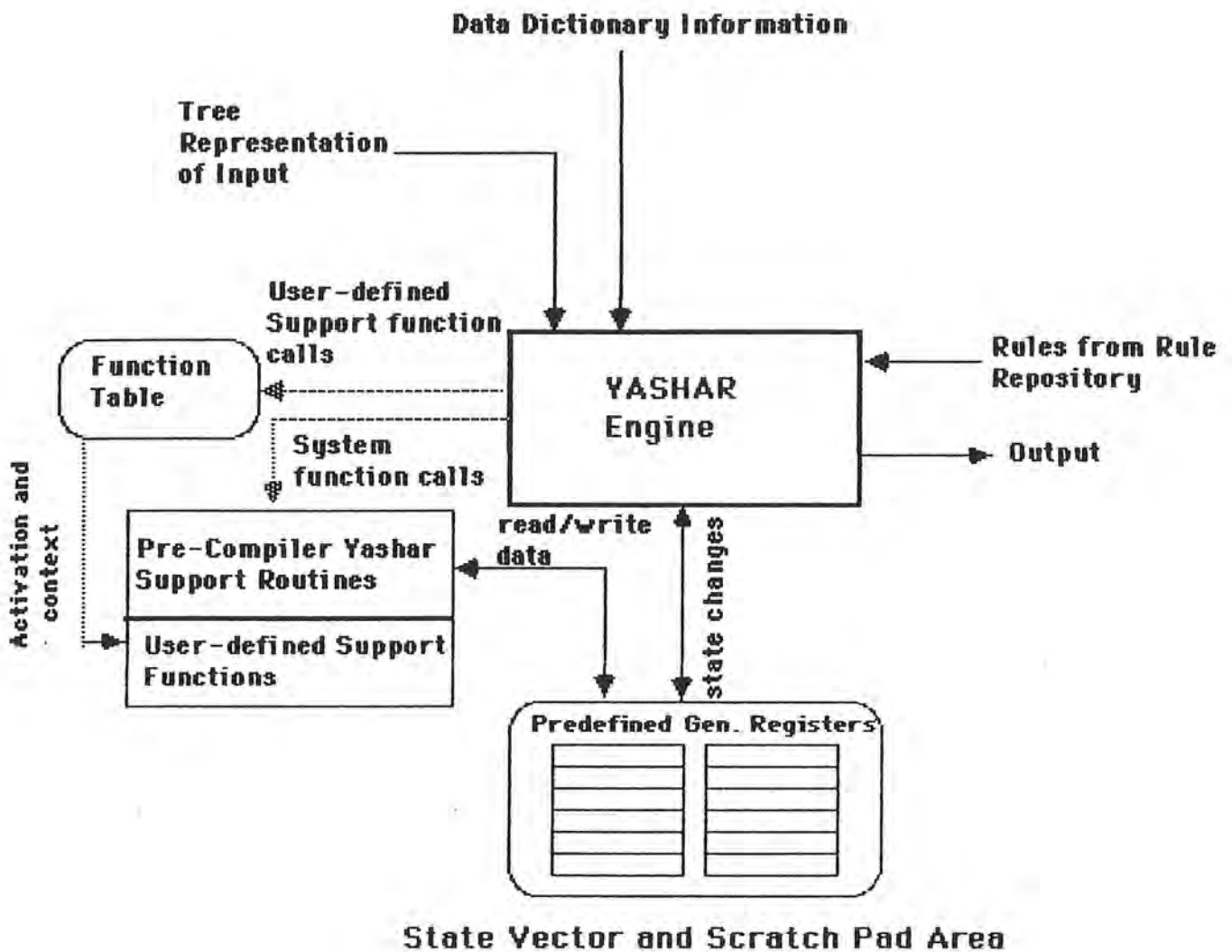
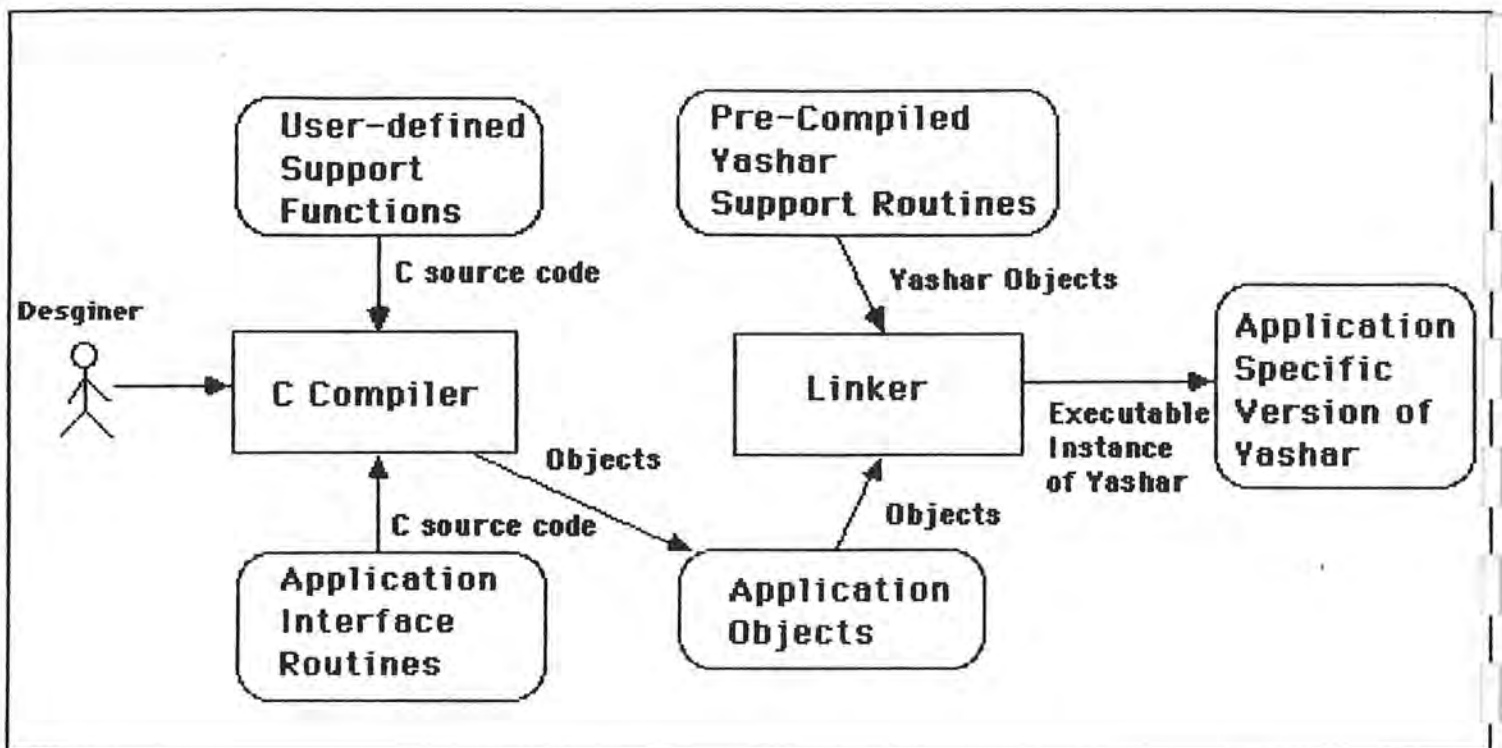


Figure 3: The Internal Tree Representation of a Modula-2 source program as it would appear in a Modula-2 to C Translator tool.



Yashar Rule Processor Execution Environment



pat10.C

```
int i,j,k,l;
```

```
main()
```

```
{  
  for ( i = 23 ; i <= 45 ; i ++ )  
  {  
    j = k;  
    l = k * k;  
  }  
}
```

pat10.mod

```
IMPLEMENTATION MODULE PAT 10;  
  VAR i,j,k,l:INTEGER;  
BEGIN  
  FOR i := 23 TO 45 DO  
    j :=k;  
    l := k*k;  
  END  
END PAT 10.
```