A graphical structure-editor that generates code for attribute grammar systems.

Daniel Ravan
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/633
NOTICE

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

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

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

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

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

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

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

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
A GRAPHICAL STRUCTURE-EDITOR
THAT GENERATE CODE FOR
ATTRIBUTE GRAMMAR SYSTEMS

by

DANIEL RAVAN

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor
Windsor, Ontario Canada 1995
THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT À LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE À LA DISPOSITION DES PERSONNE INTERESSES.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

Abstract

Attribute Grammars were proposed by Donald Knuth (1968) as a tool for describing and implementing the semantics of programming languages. The number of attribute systems now-a-days is counted by tens, some of which are truly attribute systems (such as DELTA, FNC-2, and W/AGE) and other are loosely related to the formalism (such as YACC). Some of the systems are equipped with special input languages (OLGA for FUNC-2, ALADIN for GAG and SSL for the SYNTHESIZER GENERATOR) and others accept some variant of the attribute-grammar notation as their input. In all cases, the structure of the input, although written in different syntactical forms for different systems, reflects the underlying context-free grammar and the set of semantic rules associated with it.

A graphical representation for attribute grammars, when built over an efficient recursive data structure, can be used as a base for code generation for attribute-grammar systems. To represent attribute grammars graphically, the basic constructs of the formalism must each have a corresponding graphical object that clearly shows its type and functionality.

The Attribute Grammar Generator is a visual editing tool in which objects (graphical object) can be dynamically created and linked to form an attribute grammar. The program provides facilities for creating and linking three types of primitive objects Terminals, Nonterminals and Attributes. The program also
provides several dynamic checking operations to inform the user about the consequences of the currently invoked action. The constructed attribute grammar can then be processed by a translator to generate code for previously selected tool such as W/AGE.
To my best friend Michael Tolla.
Acknowledgments

The author wishes to thank Dr. R. A. Frost for his unlimited help, encouragement, and support.
# TABLE OF CONTENTS

Abstract ........................................................................ iv  
Acknowledgments ......................................................... vii  
List of Figures ............................................................. xiv  

1 INTRODUCTION .......................................................... 1  
   ATTRIBUTE GRAMMARS .......................................... 4  
   ATTRIBUTE EVALUATION ....................................... 10  
   ATTRIBUTE GRAMMARS AS A PROGRAMMING TOOL .... 11  

Topic 1 THE HIERARCHICAL ATTRIBUTE SYSTEM ........ 12  
Topic 2 TRANSFORMATION OF PROBLEMS TO  
   ATTRIBUTE GRAMMARS ................................. 15  

2 GRAPHICAL REPRESENTATION FOR ATTRIBUTE  
   GRAMMARS ....................................................... 17  
   EXTERNAL REPRESENTATION ............................... 17  

Topic 1 CONTEXT-FREE GRAMMARS ......................... 18  
   Subtopic 1 TOKENS ........................................... 19  
   Subtopic 2 PRODUCTIONS .................................. 20  

Topic 2 SEMANTIC RULES ........................................ 23  
   Subtopic 1 ATTRIBUTES .................................... 23  

ix
Subtopic 2 FUNCTIONS ........................................ 24

3 INTERNAL REPRESENTATION FOR ATTRIBUTE

GRAMMARS ...................................................... 27

Topic 1 CONTEXT FREE GRAMMARS ......................... 28

Subtopic 1 TOKEN LIST ....................................... 28

1 TOKENS ..................................................... 29

Subtopic 2 PRODUCTION LIST ................................. 30

Subtopic 3 INSTANCES OF TOKEN ............................ 31

Subtopic 4 EXAMPLE .......................................... 32

Topic 2 SEMANTIC RULES .................................... 35

Subtopic 1 ATTRIBUTE LIST .................................. 35

1 A TOKEN’S ATTRIBUTE LIST ............................... 36

Subtopic 2 FUNCTION LIST ................................... 37

1 FUNCTIONS ................................................ 38

Subtopic 3 ARGUMENTS LIST ................................. 38

ANALYSIS OF THE INTERNAL REPRESENTATION .......... 40

4 TRANSLATION .................................................. 43

UNDERSTANDING THE STRUCTURE .......................... 43

Topic 1 NAVIGATING THROUGH THE STRUCTURE ....... 44

x
<table>
<thead>
<tr>
<th>Topic 2</th>
<th>GENERATING A GRAMMAR</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECKING</td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>Topic 1</td>
<td>DYNAMIC CHECKING</td>
<td>47</td>
</tr>
<tr>
<td>Subtopic 1</td>
<td>LEFT-RECURSION</td>
<td>47</td>
</tr>
<tr>
<td>Subtopic 2</td>
<td>TYPE CHECKING</td>
<td>50</td>
</tr>
<tr>
<td>Topic 2</td>
<td>OVERALL CHECKING</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>BUILDING A TRANSLATOR</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>WI/AGE</td>
<td>54</td>
</tr>
<tr>
<td>Topic 1</td>
<td>EXAMPLE</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>THE TRANSLATOR</td>
<td>62</td>
</tr>
<tr>
<td>Topic 1</td>
<td>TERMINALS</td>
<td>62</td>
</tr>
<tr>
<td>Topic 2</td>
<td>NONTERMINALS</td>
<td>65</td>
</tr>
<tr>
<td>Subtopic 1</td>
<td>INTERNAL REPRESENTATION OF PRODUCTIONS</td>
<td>66</td>
</tr>
<tr>
<td>Subtopic 2</td>
<td>PRODUCTION RULES</td>
<td>67</td>
</tr>
<tr>
<td>Subtopic 3</td>
<td>FUNCTION</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSION</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>APPENDIX A: USER MANUAL</td>
<td>73</td>
</tr>
<tr>
<td>Topic 1</td>
<td>WHAT IS AGG</td>
<td>73</td>
</tr>
</tbody>
</table>
Topic 2          AGG BASICS                       ........................................ 73

Subtopic 1 WORKING ON PROGRAMS ........................................ 77

1              OPEN NEW FILE                           78
2              GENERATE CODE                         79
3              EXIT AND SAVE PROGRAM                  81

Topic 3          BUILDING A PROGRAM                  81

Subtopic 1 THE BASIC COMPONENTS                     81

1              TERMINALS                              82
2              NONTERMINALS                           86

Subtopic 2 BUILDING PRODUCTIONS                     87

Subtopic 3 THE SEMANTIC EDITOR                     90

Subtopic 4 ADDING SEMANTIC RULES                   92

1              COPY AND INITIALIZE RULES              96
2              FUNCTIONS AND LIBRARIES                97

Subtopic 5 MODIFYING TOKENS                        98

1              MODIFY NAME AND PATTERN               98
2              DELETING TOKENS                      99
3              DELETING AND INSERTING INSTANCES       99

BIBLIOGRAPHY .................................................. 102
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>attribute parse tree for the number 47</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2</td>
<td>syntax tree for grammar G2</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3</td>
<td>terminals and nonterminals</td>
<td>20</td>
</tr>
<tr>
<td>Figure 4</td>
<td>a simple production</td>
<td>20</td>
</tr>
<tr>
<td>Figure 5</td>
<td>LHS and RHS of production</td>
<td>21</td>
</tr>
<tr>
<td>Figure 6</td>
<td>set of productions with the same RHS</td>
<td>21</td>
</tr>
<tr>
<td>Figure 7</td>
<td>a simple grammar</td>
<td>22</td>
</tr>
<tr>
<td>Figure 8</td>
<td>the set of attributes associated with term</td>
<td>24</td>
</tr>
<tr>
<td>Figure 9</td>
<td>selecting a production from a set of productions</td>
<td>25</td>
</tr>
<tr>
<td>Figure 10</td>
<td>the main three lists in AGG</td>
<td>28</td>
</tr>
<tr>
<td>Figure 11</td>
<td>internal representation of context-free grammar</td>
<td>34</td>
</tr>
<tr>
<td>Figure 12</td>
<td>instances of attributes</td>
<td>37</td>
</tr>
<tr>
<td>Figure 13</td>
<td>argument list</td>
<td>39</td>
</tr>
<tr>
<td>Figure 14</td>
<td>dynamic checking messages</td>
<td>41</td>
</tr>
<tr>
<td>Figure 15</td>
<td>links associated with each instance</td>
<td>44</td>
</tr>
<tr>
<td>Figure 16</td>
<td>mapping of terminal</td>
<td>63</td>
</tr>
<tr>
<td>Figure 17</td>
<td>mapping production to W/AGE-structure</td>
<td>66</td>
</tr>
<tr>
<td>Figure 18</td>
<td></td>
<td>68</td>
</tr>
<tr>
<td>Figure 19</td>
<td>mapping of function to W/AGE-function</td>
<td>70</td>
</tr>
<tr>
<td>Figure 20</td>
<td>a typical AGG cession</td>
<td>74</td>
</tr>
<tr>
<td>Figure 21</td>
<td>accelerators</td>
<td>76</td>
</tr>
<tr>
<td>Figure 22</td>
<td>setup dialog</td>
<td>77</td>
</tr>
<tr>
<td>Figure 23</td>
<td>file menu</td>
<td>78</td>
</tr>
<tr>
<td>Figure 24</td>
<td>file dialog</td>
<td>79</td>
</tr>
<tr>
<td>Figure 25</td>
<td>code generation menu</td>
<td>80</td>
</tr>
<tr>
<td>Figure 26</td>
<td>overall check messages</td>
<td>80</td>
</tr>
<tr>
<td>Figure 27</td>
<td>New token menu</td>
<td>82</td>
</tr>
<tr>
<td>Figure 28</td>
<td>new-terminal dialog</td>
<td>83</td>
</tr>
<tr>
<td>Figure 29</td>
<td>function dialog</td>
<td>84</td>
</tr>
<tr>
<td>Figure 30</td>
<td>new attribute dialog</td>
<td>85</td>
</tr>
<tr>
<td>Figure 31</td>
<td>attribute data-type dialog</td>
<td>85</td>
</tr>
<tr>
<td>Figure 32</td>
<td>new-nonterminal dialog</td>
<td>86</td>
</tr>
<tr>
<td>Figure 33</td>
<td>new nonterminal</td>
<td>87</td>
</tr>
<tr>
<td>Figure 34</td>
<td>token-area scrolls</td>
<td>87</td>
</tr>
<tr>
<td>Figure 35</td>
<td>production right hand side</td>
<td>88</td>
</tr>
<tr>
<td>Figure 36</td>
<td>adding new production-line</td>
<td>89</td>
</tr>
<tr>
<td>Figure 37</td>
<td>hide selected production</td>
<td>89</td>
</tr>
<tr>
<td>Figure 38</td>
<td>attribute menu</td>
<td>90</td>
</tr>
<tr>
<td>Figure 39</td>
<td>the semantic editor</td>
<td>91</td>
</tr>
<tr>
<td>Figure 40</td>
<td>control area push-buttons</td>
<td>92</td>
</tr>
</tbody>
</table>
Figure 41  selecting attribute .................................. 94
Figure 42  Function dialog ..................................... 95
Figure 43  New-Function dialog ................................. 97
Figure 44  modify and delete tokens ............................ 98
Figure 45  left-recursion ...................................... 100
Figure 46  deleting instances ................................. 100
Figure 47  insertion of instance ............................... 101
Chapter 1 INTRODUCTION

In the early sixties a question arose within the Computer Science community: how should we define the meaning of statements in algorithmic languages? A mechanism proposed in 1960 by Ned Iron stated: if we know the meaning of $A$ and $B$ then we know the meaning of $A+B$, $A-B$, $A*B$, $A/B$, and $(A)$. The meaning of large expressions can then be recursively built by synthesizing the meanings of its sub-expressions. Alblas [1]. In 1968 Donald E. Knuth[9] introduced Attribute Grammars as a tool by means of which the semantics of programming languages can be described and implemented. The semantics of a language is described by means of attributes attached to the constructs of the grammar (context-free grammar) representing the language. Each attribute is defined over a well-defined range, the value of any language expression can be obtained by applying a set of semantic equations associated with the grammar production rules.

When first introduced, attribute grammars found some use in compiler work, however research has shown that the formalism is much more powerful and need not be restricted to one type of application. The power of attribute grammars, when unleashed, can cover almost every aspect of computer science. Papers in the filed are appearing more frequently than ever, introducing and investigating attribute systems and new uses for the formalism, the last of which is the attribute-grammar programming paradigm.
The number of attribute systems now-a-days is counted by tens, some of which are truly attribute systems (such as DELTA, FNC-2, and W/AGE) and other are loosely related to the formalism (such as YACC). Some of the systems are equipped with special input languages (OLGA for FUNC-2, ALADIN for GAG and SSL for the SYNTHESIZER GENERATOR) and others accept some variant of the attribute-grammar notation as their input. In all cases, the structure of the input, although written in different syntactical forms for different systems, reflects the underlying context-free grammar and the set of semantic rules associated with it. The following, for example, is a subset of a disk-calculator specification written for the SYNTHESIZER GENERATOR:

```
exp : Null [* : "<exp>" ]
  | Plus [* : "(" @ "+" @ ")")
  | Minus [* : "(" @ "-" @ ")")
  | Times [* : "(" @ "*" @ ")")
  | Uminus [* : "+" @ "]
```

The notation looks a little different from the well-known notation of attribute- grammars, however, the structure of the input is very much the same. Notice that the syntax is rich with special symbols (a common phenomenon in attribute- grammar systems) which makes it error prone and in many cases hard to use. The attribute-grammar programming paradigm is relatively new, and has not yet had the proper chance to demonstrate its full capabilities. Users often shy away from attribute systems because of their cryptic notation, and because they require fundamental changes to the common method of writing and reasoning.
about programs. A careful examination to the set of productions in the example above will show that, regardless of their odd appearance, the special symbols seem to follow a fixed pattern that can be easily anticipated and automatically generated.

Reducing the "distance" between the user and the computer is one area in computer science that has been very active since the introduction of the first computers. In the early days, the only means available to the user to interact with computers, was a set of binary switches (part of the computer hardware) each of which represent an entry of zero or one. Not very long after that, computer scientists introduced the first high-level programming language which allowed users to specify problems in mathematical-like notation. Throughout the last thirty years programming languages improved significantly. The introduction of new languages and new paradigms, such as functional, object oriented, and recently the attribute-grammar programming paradigm, has revolutionized the method of writing and reasoning about programs. When combined together, the attribute grammar and functional programming paradigms, form a powerful programming environment in which programs are executed on the specification level. W/AGE is such a system where specifications can be analyzed and executed, a significant improvement over any available high-level programming language. Our work is another step toward higher-level programming environments, we have provided a method for graphical and structural representation of attribute grammars and proved by implementing the system that such representation can be used as a
base for code generation for attribute systems.

Our investigation of systems like W/AGE and YACC lead us to the conclusion that input to these systems can be automatically generated by applying a simple translation mechanism to the original attribute grammar. If the well-known attribute-grammar notation is represented by some dynamic and recursive data structure, the translation can then be viewed as a simple mapping applied to elements of a recursively-traversed structure. If the structured attribute-grammar is maintained and tested carefully throughout the construction process, the code generated by a successful translation process is guaranteed to be syntactically correct. Automatic code generation for attribute-grammar systems, will very much simplify the process of coding. In the following chapters, we introduce a method for graphically representing attribute grammars, and a data structure in which attribute grammars can be stored, mentioned and tested. We will also introduce our own implementation of the suggested method together with a step by step instruction on the construction of translators using our implementation.

**Section 1 ATTRIBUTE GRAMMARS**

Attribute Grammars are based on Context Free Grammars which can be defined as a tuple:

\[ G = \{ N, T, P, S \} \]

where

- \( N \) = finite set of Nonterminals.
- \( T \) = finite set of Terminals.
- \( P \) = finite set Production rules
\( S = \text{starting symbol (initial nonterminal)} \)

Two finite disjoint sets of attributes are associated with each nonterminal, one is referred to as synthesized and the second as inherited. Synthesized attributes, as defined by Knuth[10] are "aspects of meaning which are built up from within the phrase", while inherited attributes are "aspects of meaning which come from the context of a phrase". Synthesized attributes are propagated upward the parse tree and inherited attributes downward. Productions in attribute grammars are associated with a finite set of semantic rules used to evaluate attributes for production symbols. Terminals may have only synthesized attributes while nonterminals may have both types of attributes. However, in many formalisms the initial nonterminal may have only synthesized attribute (this condition is not always imposed Alblas[1], in some cases start symbols inherit the program environment). Attributes are attached to their grammar symbols (terminals and nonterminals) in the derivation tree, which is constructed by the underlying context-free grammar. As introduced by Knuth in his original paper, attributes can all be synthesized, the attribute grammar is then called purely synthesized (this can play an important role in attribute evaluation) or all inherited, the attribute grammar is then called purely inherited. A derivation tree with all attribute instances having values, is referred to as a decorated tree. The value of an attribute instance may depend on one or more values of other attributes, and is obtained by applying one or more semantic rules to them. In some cases the attribute value
may depend directly or indirectly on its itself, i.e. the attribute on the left-hand side of the semantic-rule appears directly or indirectly in the right-hand side of the same rule. This type of attribute dependency causes what is referred to as circularity, which must be detected and removed before starting the evaluation process. Mechanisms for the circularity check have been proposed by several researchers, most of which are built as part of an attribute evaluation system.

When evaluation starts, all attributes except the inherited attribute of the start symbol, if any, and the synthesized of the terminals are unavailable. The start symbol inherits the program environment, while terminals have values determined by the lexical scanner [1].

The following grammar (G1) defines the syntax of a language that consists of numbers between 0 and 99.

```
grammar G1
Number  ::=  One_digit | Two_digits
Two_digits ::=  First_digit One_digit
First_digit ::=  1 | 2 | ..... | 9
One_digit  ::=  0 | First_digit
```

The nonterminal `One_digit` excludes numbers like 00, 01, ..., 09.

To construct an attribute grammar for G1, we must associate a set of attributes with the grammar symbols and a set of evaluation rules with the productions.

For G1 we need only one synthesized attribute, we refer to it as `VAL`, it is synthesized because the only flow of information in the parse tree of this specific example is, bottom up. The set of attributes associated with G1 is as follows:
Number: VAL↑↑

One_digit: VAL↑↑

First_digit: VAL↑↑

Two_digits: VAL↑↑

Where ↑ stands for synthesized attribute. The corresponding attribute grammar for G1 is as follows:

Number ::= One_digit

\[
\begin{align*}
\text{Number}\downarrow \text{VAL} & := \text{One_digit}\downarrow \text{VAL} \\
& \mid \text{Two_digits} \\
\text{Number}\downarrow \text{VAL} & := \text{Two_digits}\downarrow \text{VAL}
\end{align*}
\]

Two_digits ::= First_digit One_digit

\[
\begin{align*}
\text{Two_digits}\downarrow \text{VAL} & := \text{First_digit}\downarrow \text{VAL} \ast 10 + \text{One_digit}\downarrow \text{VAL}
\end{align*}
\]

One_digit ::= '0'

\[
\begin{align*}
\text{One_digit}\downarrow \text{VAL} & := 0 \\
& \mid \text{First_digit} \\
\text{One_digit}\downarrow \text{VAL} & := \text{First_digit}\downarrow \text{VAL}
\end{align*}
\]

First_digit ::= '1'

\[
\begin{align*}
\text{First_digit}\downarrow \text{VAL} & := 1 \\
& \mid \text{One_digit} \\
& \mid \text{Two_digits} \\
& \mid \text{One_digit}\downarrow \text{VAL} \ast 10 + \text{One_digit}\downarrow \text{VAL}
\end{align*}
\]
First_digit VAL := 9

Notice that each alternative on the right hand side of a production has only one semantic equation, this is owing to the fact that there is only one attribute. The general rule is, a semantic equation must be supplied for each synthesized attribute appearing on the left-hand side of the production and for each inherited attribute appearing on the right-hand side.

Figure 1 attribute parse tree for the number 47

Number=47

Two_digits=47

First_digit=4 One_digit=7

4 7

For simple grammars like G1, attribute-evaluation is a simple straightforward process. Complicated grammars, however, require complicated preparation and calculation, there are cases when evaluation is an infinite non-terminated process (circularity). Evaluators usually check for circularity or assume its absence (because of the exponential complexity of the circularity-testing algorithm) before starting the evaluation process. Below is an example of a complex flow of attributes.

grammar G2
s ::= XYZ
X ::= x
Y ::= y
S : A | B | C | D | E | F | G | H

X := x

Y := y

Z := z

The flow of information in this grammar is not as easy as it is in G1, notice that there are four semantic equations associated with first production.

The numbers on the nodes of the syntax tree below represent the order of attribute evaluation, which is determined by attribute dependencies.
The order of evaluation, as shown in the tree, is $S \rightarrow A, Z \rightarrow H, Z \rightarrow G, X \rightarrow C, X \rightarrow D, S \rightarrow B, Y \rightarrow E$ and $Y \rightarrow F$. Where $S \rightarrow A$ is the initial value of the start symbol. An overview of attribute grammar theory and implementation is given in Courcelle[3].

**Section 2 ATTRIBUTE EVALUATION**

An attribute evaluator is a program which, when executed, computes values of attribute instances by applying the corresponding attribute equations in some order. The order in which attributes are evaluated depends on many factors, and is restricted by some rules, such as, arguments of evaluation instruction must be available before the instruction is called (this might not be the case when working with lazy evaluation). The main task of the evaluator is to compute the value of unknown attributes to produce a decorated tree, a process sometimes referred to as tree decoration. Attribute evaluators may differ in their adopted method and the type of restriction imposed on the input, but their main goal always stays the same. There are two main types of attribute evaluators, **dynamic** and **static** Alblas[2]. Static evaluators, construct dependency paths for the evaluation process. Joined
together, dependency paths form a dependency graph. Every derivation tree has a dependency graph associated with it. Nodes in a dependency path form a list in which every element depends on the sub-list of elements that proceed its position in the list. A dynamic evaluator constructs a dependency graph from the input (a source program and its associated syntax tree) and then topologically sorts the nodes of the graph to determine the order of evaluation, i.e. the evaluation order is determined at the attribute evaluation time (the time when the source is compiled), as a consequence different trees of the same grammar require different evaluation orders. Static evaluators construct no dependency graph, because the order of evaluation is already determined by the evaluator generator at the evaluator generation time. In lazy evaluators, such as W/AGE Frost[5], the order of evaluation is determined at execution time when attributes are only evaluated when required and only to the extent required.

Section 3 ATTRIBUTE GRAMMARS AS A PROGRAMMING TOOL

Informal specification of problems often leads to different interpretation by different individuals. Formal specification, on the other hand, avoids ambiguity and provides the means to express problems very clearly to ensure that all developed solutions address the same issue. Hehner[6] suggested the use of grammar as a formal method of specifying programs, "a particular grammar shows a particular way of structuring the problem that can serve as a model for
the solution"[6]. When grammars are used as programs, the compiler must be a parser generator which can supply algorithms, such as LALR(1). When Hehner discussed this approach, he mentioned very briefly the importance of adding attributes as a semantic tool. Prior to Hehner, many researchers introduced the concept of formally specifying programs, such as Katayama and Mayor.

**THE HIERARCHICAL ATTRIBUTE SYSTEM**

A hierarchical decomposition of large programs to their concrete executable constructs, proved to be a very successful method in constructing large and complex software. The basic abstraction which cannot be further decomposed are referred to as modules. Each module perform a specific task according to the method of decomposition. The program itself is a sequence of calls (in some order) to these modules. In general modules act on some input data to produce some output. The output can be used as input for some other module or as a final result of problem. The similarity between this approach and attribute grammar is clear. modules can easily looked upon as nonterminals, their input and output are synthesized and inherited attributes, the program itself can be viewed as an attribute grammar. This approach was first introduced in 1981 by Katayama[8]. Using attribute grammars Katayama suggested a Hierarchical and Functional Programming method (HFP), in which a program is decomposed into abstract modules (nonterminals) and coded as functions using any available
purely functional programming language. The method of decomposition here
 corresponds to production rules, computation in the modules corresponds to
 semantic rules, the whole function of the program is to evaluate attributes. The
 use a functional programming languages is essential to this approach because
 they insure the uniqueness and independence of the modules regardless of their
 position in the program, which play a very important role in the process of attribute
 evaluation.

 According to Katayama, the final program can be viewed as a tree or number
 of trees (if the grammar is ambiguous), modules are nodes of the tree, input and
 output are attributes that propagate upward and downward in the tree. To illustrate
 Katayama’s method, consider a program that calculates the \( n \)th Fibonacci number.
 Using any imperative programming language that supports recursion the program
 can be easily coded as a single function:

 ```c
 int fib(int n) {
     if (x < 2)
         return(1);
     else
         return(fib(n-2) + fib(n-1));
 } /*fib*/
```

 Using the new approach we first decompose the program into modules \( \text{fib}_1 \text{ fib}_2 \)
 ... \( \text{fib}_m \) denoted as:

 ```
 $\text{fib} \ast \text{fib}_1 \text{ fib}_2 ... \text{fib}_m$

```

 The decomposition is governed by a set of decomposition rules, which can be
 straightforward and limited in number for simple programs. In [8], Katayama
suggested the following decomposition for the Fibonacci problem above:

\[
\begin{align*}
\text{fib} & : \text{fib1 fib2 where I.fib} \geq 2 \\
\text{I.fib} & = n \\
\text{I.fib1} & = \text{I.fib} - 1 \\
\text{I.fib2} & = \text{I.fib} - 2 \\
\text{S.fib} & = \text{S.fib1} + \text{S.fib2} \\
\text{fib} & \quad \text{where I.fib} < 2 \\
\text{I.fib} & = n \\
\text{S.fib} & = 1
\end{align*}
\]

Where \( I \) (inherited attribute) is the input, \( S \) (Synthesized attribute) is the output, and \( n \) is the order of the Fibonacci number we wish to compute.

The program now can be viewed as an attribute tree, the problem has been transformed into a new one, that is, decorating an attribute tree, a task that can be efficiently accomplished by the lazy evaluation mechanism provided by the underlying functional programming language. A similar approach to that of Katayama's introduced by Mayor[11], however the last deals with Logic Programming. The idea of using attribute grammars as a programming tool is not new, it started when Knuth introduced the concept. We quote the following from Knuth[10], "it is a happy circumstance when an intuitive description of a system can be almost automatically transformed into practical working model based on that system". There are many advantages associated with the use of attribute grammar as a programming language. The formalism can be easily viewed as a high-level programming language, it is non-procedural, and allows hierarchical development of programs, it also can be tailored to any special requirement and there exist many efficient attribute evaluation systems.
TRANSFORMATION OF PROBLEMS TO ATTRIBUTE GRAMMARS

Katayama's system assumes that all atomic executable modules are attached to the nodes of a computation tree (decomposition tree), which must be constructed according to some appropriate decomposition rules (in the case of Fibonacci number, testing the value of the inherited attribute \( f \)). This approach, although using attributes, cannot be considered as a full implementation of attribute grammars, because the parsing process is completely eliminated and replaced by decomposition rules that depends on something other than the input.

One of the major problems in the attribute-grammar programming paradigm is how to express a problem such as generating Fibonacci numbers using an attribute grammar, i.e. how to relate the structure of the tree to the structure of the input string (in this case a decimal number). This is one of the reasons why many attribute systems, assume the existence of the parse tree and restrict their functionality on attribute evaluation. \( W/AGE \) (see section 5.1) was the first system with both syntactic and semantic capabilities for building executable specification using higher order functions. Introduced in 1990 by R. A. Frost, \( W/AGE \) comes with new prospective for viewing problems and introduced a new technique for modifying problems (such as generating Fibonacci number) to be described completely by an attribute grammar Frost[4]. To illustrate, consider the following attribute grammar:
number ::= element

number|FIB = 1

| elm elm

number|FIB = 1

number|PFIB = 1

| elm number1

number|FIB = number1|FIB + number1|PFIB

number|PFIB = number1|FIB

Where \textit{FIB} stands for Fibonacci value of the number, \textit{PFIB} stands for the previous Fibonacci value of the number, and \textit{element} is a recognizer for an object, i.e. the grammar assumes that the input is a list of objects each of which represents one unit in the number. For example, "11111" will be recognized by the grammar as 5 (assuming that \textit{element} is defined as a recognizer for the character '1').
Chapter 2  GRAPHICAL REPRESENTATION FOR ATTRIBUTE GRAMMARS

Large amounts of information can often be conveyed to users in a short period of time using a graphical representation of objects. Thousands of numbers can be fit into one simple graph from which users can build a general concept, about the phenomenon under investigation, in one glance. Being self-explanatory and less symbolic, graphical objects are easy to work with and provide an interesting working environment.

Section 1 EXTERNAL REPRESENTATION

In order to represent attribute grammars graphically, the basic constructs of the formalism must each have a corresponding graphical object that clearly shows its type and functionality. Most of the attribute grammar constructs can be categorized into more than one type. Tokens, for example, can be terminals or nonterminals, attributes can be synthesized or inherited. Such properties must be allowed to surface in the corresponding object to provide a clear picture of the formalism. The functionality of attribute-grammar constructs ranges between simple uninterrupted symbols to a sophisticated equation that combines the results of a set of other equations associated with other symbols. Such nontrivial functionality is much better expressed to the users in parts, to avoid any confusion that may arise when
all the details are packed in a single complicated graphical object, that looks more
like a maze than a set of linked objects.

Before suggesting any specific graphical image for the formalism, we develop
a list of constructs that are to be presented and their possible set of types. The
main two components in any attribute grammar are:

1. The underlying context-free grammar.
2. The set of semantic equations.

Each of the two components above consists of a set of simpler objects linked
together to form a single structured object. The second, although it has its own
structure, is completely dependent on the first. It consists of a set of functions
each of which has a name and a set of links to objects referred to as arguments.

CONTEXT-FREE GRAMMARS

A graphical representation of attribute grammars must provide sufficient
information about the (attribute grammars) components and the languages they
represent. A context-free grammar, as defined earlier, is a tuple of four:

\[ G = \{ \mathit{N}, \mathit{T}, \mathit{S}, \mathit{P} \} \]

where

\[ \mathit{N} = \text{finite set of nonterminals.} \]

\[ \mathit{T} = \text{finite set of terminals.} \]

\[ \mathit{S} = \text{Starting symbol } \{ \text{initial nonterminal } \}. \]
P = finite set of production rules.

Consider the following context-free grammar:

\[
\begin{align*}
\text{Grammar G2} \\
N &= \{ \text{exp, term, factor} \} \\
T &= \{ (, ), \text{, , *, /, number} \} \\
S &= \{ \text{exp} \} \\
P &= \{ \\
&\quad (1) \text{ exp} \to \text{ term } \cdot \text{ exp} , \\
&\quad (2) \text{ exp} \to \text{ term } \cdot \text{ exp} , \\
&\quad (3) \text{ exp} \to \text{ term} , \\
&\quad (4) \text{ term} \to \text{ factor } \cdot \text{ term} , \\
&\quad (5) \text{ term} \to \text{ factor } / \text{ term} , \\
&\quad (6) \text{ term} \to \text{ factor} , \\
&\quad (7) \text{ factor} \to (\text{ exp}) , \\
&\quad (8) \text{ factor} \to \text{ number} \\
\}
\end{align*}
\]

Assuming that the symbol \textit{number} represents a construct that directly recognizes numbers, the above is a definition of a simple context-free grammar specifying the syntax of a language that consists entirely of simple mathematical expressions.

\textbf{TOKENS} Since terminals and nonterminals are the building blocks of any context-free grammar we start by providing atomic graphical constructs that distinctly represent each and every one of them. The suggested graphical construct is a simple box with the name of the token (token refers to terminal and nonterminal) inside. The size of the box depends on the length of the token name and the currently selected font. To separate terminals from nonterminals, different types of lines are used to draw the borders of the boxes. In the current implementation
of AGG, we used *solid* lines for terminals and *dashed* lines for nonterminals.

Figure 3 terminals and nonterminals

\[
\begin{align*}
\text{exp} & \quad \text{number} \quad \star \quad \text{-} \\
\text{term} & \quad ( \quad + \\
\text{factor} & \quad ) \quad / \\
\text{nonterminals} & \quad \text{terminals}
\end{align*}
\]

**PRODUCTIONS** Productions are non-atomic objects, each of which consists of a nonterminal in the left-hand side and a set of tokens to the right-hand side. To represent productions graphically we choose to link the atomic graphical constructs corresponding to each token in the same order they appear in the production:

Figure 4 a simple production

\[
\begin{align*}
\text{exp} \quad & \text{term} \quad + \quad \text{exp}
\end{align*}
\]

Although it is clear that the above graphical object represents the production \((\text{exp} :\text{term} + \text{exp})\), it is not clear that the structure consists of two parts, *left-hand side* and *right-hand side*. By slightly modifying the link, we succeeded in providing a more precise representation and a base for more efficient representations for productions that share the same left-hand side symbol:
exp

  term + exp

Following the same style, productions 1, 2, and 3 can all be represented by a single graphical object:

exp

  term + exp

  term - exp

  term

Putting all the productions together the whole grammar can be presented as follows:
Figure 7: A simple grammar

```
exp
  term + exp
  term - exp
  term

term
  factor

factor
  * term
  / term
  factor

number
```

All terminals, nonterminals, and productions are clearly represented in Figure 7 above. However, one can easily recognize two disadvantages:

1. It is not obvious which of the nonterminals is the initial nonterminal.
2. For multiple-production attribute grammars, the picture can spread across a wide area, which makes the overall representation incomprehensible.

The second can be easily solved by introducing a new higher level graphical representation, which can help in preserving space and reducing complexity. The new level makes use of the facilities provided by the underlying windowing environment (in the current implementation X Windows). The whole grammar can be presented by a set of graphical objects (push buttons, labeled with the token name) divided in two separate groups. The first represents nonterminals and the second represents terminals, the start symbol can be characterized by giving its
object a distinct color. In any windowing-environments objects (such as pushbuttons) are sensitive to user action, and can be set to invoke a desired function as a response to a given user action (such as clicking the mouse button while its pointer positioned over the given push button). In the current implementation, dragging a nonterminal object and dropping it on the drawing area, will result in drawing all of the productions associated with it (see user manual).

SEMANTIC RULES

A semantic rule is an object that has a name and a set of arguments associated with it. An argument is an attribute associated with a token (an argument can be any object of the right type, however, to reduce the complexity of the structure and to speedup the checking process, we restrict arguments in the current implementation to attributes). The building components of semantic rules are attributes (excluding the cases when initialize rules are used).

ATTRIBUTES  To represent attributes, we chose the same boxes we used for tokens, however, these boxes are visible only when the user positions the mouse pointer over the required token and presses on the mouse button, when the button is released, attributes become invisible again. Like terminals and nonterminals, we defined inherited attribute boxes to be of solid borders (because terminals are associate only with inherited attributes), and synthesized attributes as boxes
with dashed borders. To illustrate, suppose that the following two attributes are associated with the nonterminal \textit{term}: 

\begin{verbatim}
VALUE
ENVIRONMENT
\end{verbatim}

Clicking on the mouse button while the pointer is on the box labelled \textit{term} (when the program is in \textit{edit} mode) will result in listing all the attributes associated with the token, each of which in a separate box figure (8):

Figure 8 the set of attributes associated with \textit{term}

\begin{verbatim}
before user presses on mouse left button
\begin{verbatim}
expr
\begin{verbatim}
\wedge term + expr
\end{verbatim}
\begin{verbatim}
\wedge term + expr
\end{verbatim}
\begin{verbatim}
VALUE \leftarrow synthesized attribute
\end{verbatim}
\begin{verbatim}
ENVIRONMENT \leftarrow inherited attribute
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}

\textbf{FUNCTIONS} An attribute value can be the return values of an apply rule (the result of applying a function to other attribute values), or a copy of another attribute value, or initialized to some constant. Since there is no limit to the number of attributes associated with each token, and no limit to the number of arguments associated with each function, representing semantics rules with a set of graphical links between attributes boxes can be quite confusing, specially for tokens with
large number of attributes. One possible solution to this problem (implemented in AGG) is to allow multiple-level representations for each production.

When a set of productions (such as the one in figure 6) is selected, the user can choose any single production by selecting any of its right-hand side components. Once a production is selected (see figure 9) the user can select any of its components to view the set of attributes associated with it.

Figure 9 selecting a production from a set of productions

\[
\begin{align*}
\text{expr} \\
\text{\& term} & \quad - \quad \text{expr} \\
\text{VALUE} \\
\text{ENVIRONMENT}
\end{align*}
\]

When an attribute is selected, the associated function (or constant if any) will be displayed on a dedicated area using the following format:

\[
\text{attribute} = \text{function} \_ \text{name} (\text{arg} \_ \text{list}) \\
\text{where} \\
\text{attribute} :: \\
\text{token} \_ \text{name} \text{token} \_ \text{number} \rightarrow \$\text{dir} \_ \text{e} \_ \text{ion} \text{attribute} \_ \text{name} \\
\text{token} \_ \text{number} :: \\
\text{the order of the token in the production (lhs = 50).} \\
\text{dir} \_ \text{e} \_ \text{ion} :: \\
\text{'}u\text{' for synthesized} \\
\text{\| \text{'}d\text{' for inherited.} \\
\text{arg} \_ \text{list} :: \\
\text{list of arguments associated with the function} \\
\text{each of which represents an attribute.}
\]

25
The following, for example, shows that the value of the synthesized attribute $VAL$. associated with the token $exp$ in the left hand side, is computed by applying the function $sub$ on the synthesized attribute $VAL$ associated with tokens $term$ and $exp$ on the right hand side of the production:

$$exp \, S0 \rightarrow \_u \, VAL = sub(term \, S1 \rightarrow \_u \, VAL, \ exp \, S2 \rightarrow \_u \, VAL)$$

$S0$, $S1$, $S2$ etc. are used to identify different instances of the same token. When $copy$ rule is used the format is as follows:

$$attribute = attribute$$

When $initialize$ rule is used the format is as follows:

$$attribute = initial\_value$$
where
$$initial\_value = any\ constant\ value$$
Chapter 3  INTERNAL REPRESENTATION
FOR ATTRIBUTE GRAMMARS

An attribute grammar, as viewed by AGG, is a structure of three-components
each of which is a list of linked objects. These lists are dynamically structured
while the attribute grammar is being constructed, figur(10). To make these lists
globally accessible, their addresses are included in the global structure PgmInfo
of type ProgramInformationStruct.

To access AGG’s data structure, external modules must include the following
two lines:

```c
#include "agg.h"
extern ProgramInformationStruct PgmInfo;
```

Having these two lines included, a translator can access the main three lists as
follows:

1. **Attribute list**: PgmInfo.AttList
2. **Function list**: PgmInfo.FunList
3. **Token list**: PgmInfo.Program
Figure 10 the main three lists in AGG

attribute list function list
token list

Attribute-grammar as viewed by AGG

CONTEXT FREE GRAMMARS

TOKEN LIST  Terminals and nonterminals in AGG are grouped in one single list of type AggProgram. The type AggProgram is defined in AGG as follows:

```c
typedef struct
    AggToken *first, *last;
    AggToken *selected;
    AggToken *start_symbol;
    int prd_count;
    int term_count;
    ]AggProgram;
```

where
first  first token in the program.
last  last token in the program.
selected reserved for AGG use.
start_symbol current initial nonterminal.
prd_count number of nonterminals in the grammar.
term_count number of terminals in the grammar.
To make the list globally available we included its address in the global structure 
_PgmInfo_. Once the external variable _PgmInfo_ defined in a module, the list com-
ponents can be easily accessed via the data member _Program_, i.e., _PgmInfo.Program_.

**TOKENS** Terminals and nonterminals in AGG are instances of the same type

_AggToken_. The type _AggToken_ is define in AGG as follows:

```c
typedef struct token AggToken;
struct token{
    AggElement *root;
    AggList *attribute;
    Widget widget;
    int visible;
    char *pattern;
    char type;
    AggElement *first_instance, *last_instance;
    AggProductionLine *first_line, *last_line;
    AggToken *next, *previous;
    int order;
}; /*token*/
```

where

- root: an instance of the token, the type _AggElement_.
- attribute: list of attributes associated with the token.
- widget: reserved for AGG use.
- visible: reserved for AGG use.
- pattern: a string pattern associated with token terminals only), set to NULL when token type is NONTERMINAL.
- type: one of the predefined constant TERMINAL or NONTERMINAL.
- first_instance: points to the first instance of the token, see the type _AggElement_.
- last_instance: points to the last instance of the token, see the type _AggElement_.
- first_line: points to the first production associated with the token (nonterminal), see the type _AggProductionLine_.
- last_line: points to the last production associated with the token (nonterminal), see the type _AggProductionLine_.
- next: points to the next token in the token list.
- previous: points to the previous token in the token list.
**AggToken** is a complicated data structure, it contains all the necessary information to define an attribute-grammar construct. The type of each token is determined by the data member *type* which is set to one of the predefined constants **TERMINAL** or **NONTERMINAL**. AGG uses Tokens as template for creating instances of type *AggElement* each of which represent an occurrence of the token in the context-free grammar. Instances of a token are grouped in a list whose address assigned to the data member *first_instance*. The address of the last element is assigned to the data member *last_instance*.

**PRODUCTION LIST**  A nonterminal in a grammar can sometimes be expanded in many different ways, each of which is presented by a production whose left hand side symbol is the same nonterminal. In AGG, productions that share the same left hand side symbol are grouped in one list of type *AggProductionLine*, each production line consists of an instance of the left hand side symbol and a list of instances each of which represent one of the right hand side symbols. The type *AggProductionLine* is defined in AGG as follows:

```c
struct production_line{
    AggElement *lhs;
    AggElement *first, *last;
    AggElement *selected;
    AggToken *parent;
    struct production_line *next, *previous;
}; /*production_line*/
typedef struct production_line AggProductionLine;
```

where

*lhs*  an instance of token representing the left-hand side of production.
first  points to an instance of the first token in the
1st-hand side of the produion.
last   points to an instance of the last token in the
1st-hand side of the produion.
solved reserved for AGG use.
parent points to the token representing the lef-hand
side of the produion.
nexy   points to the next produion in the list.
previous points to the previous produion in the list.

Given the following production:

    exp := term plus exp

The corresponding AggregationLine members will be set to the following
values:

    line->lhs   ::= points to an instance of exp.
    line->last  ::= points to an instance of exp.
    line->first ::= points to an instance of term.
    line->parent:: points to token exp.

INSTANCES OF TOKEN

An instance in AGG is a structure of type AggElement. Each instance in a
production line corresponds to an occurrence of a symbol in the grammar. The
type AggElement is defined in AGG as follows:

    typedef struct element AggElement;
    struct element{
        int status;
        char *name;
        char type;
        AggRect rect;
        AggAttribute *attribute;
        AggAttribute *last_att;
        int att_count;
        struct element *next, *previous;
        struct element *left, *right;
        AggProductionLine *parent_prd_line;
        AggToken           *parent_token;
        int order;
    }/*element*/
where
status  set to the predefined constant ON if the instance
        is scheduled for deletion.
name   instance name.
type   instance type, one of the predefined constants
        TERMINAL or NONTERMINAL.
re     reserved for AGG use.
attribute list of attributes associated with the instance,
        see the type AAttribute.
last_att points to the last attribute associated with the
        instance, see the type AAttribute.
att_count number of attributes associated with the instance.
next    next instance in the list of instances
        associated with production.
previous previous instance in the list of instances
        associated with production.
right   next instance in the list of instances associated
        with token.
left    previous instance in the list of instances
        associated with token.
parent_prd_line points to the production containing the instance.
order   an integer representing the order of the instance
        in the production, starting from the lefthand side
        the first instance assigned the order (·1).

instances are added to list in the same order they appear in the production. Each
instance of a token represents a cross-point between two lists, a list of instances
associated with a token and a list of instances associated with production. The
first can be traversed using the data members left, right, and the second using
previous, next.

EXAMPLE    Figure (12) shows the status of the program structure before and
after adding each of the first three productions of grammar G2 (section 2.1). For
every new token (terminal or nonterminal) defined by the user, AGG attaches a
new instance of type AaggToken to the token list. Figure (12.1) shows the list after
adding all the tokens associated with grammar. Figure (12.2) shows the structure of the list after adding the production \(\text{exp} \rightarrow \text{term} + \text{factor}\).

Productions are defined by their left hand side symbols, adding new production to the grammar will reflects the following changes on the program structure:

1. A new instance of the type \textit{AggProductionLine} will be attached to the list of productions associated with the token representing the left hand side symbol of the production.

2. A new instance of the same token will be attached to the list of instances associated with token. The address of the same instance will be assigned to the \textit{lhs} data member of the newly created production (see \textit{prdI} in figure 12.2).

The right-hand side of the production consists of a set of symbols each of which must associate with a predefined token in the tokens list. For every symbol in the production, AGG creates an instance of the corresponding token and links it to the list of instances associated with token and to the list of instances associated with the production. Figure (12.2) shows the internal structure of the program after adding production (1) of grammar \textbf{G2}. Similarly production (2) and (3) are added to the structure (see figure 12.4 ).
Figure 11 internal representation of context-free grammar
SEMANTIC RULES

Semantic rules are represented by a second layer of data structure and graphical objects. In most cases these objects are partially visible to allow for more comprehensible representation of the attribute grammar.

ATTRIBUTE LIST All attributes associated with a given grammar are grouped in one list of type \texttt{AggList} whose elements are structures of type \texttt{AggListElement} figure (10). The type \texttt{AggList} is defined in AGG as follows:

\begin{verbatim}
typedef struct{
    AggListElement *list, *last;
    int count;
}AggList;
\end{verbatim}

Each element in the list corresponds to one attribute declared in the program. The type \texttt{AggListElement} is defined in AGG as follows:

\begin{verbatim}
struct list(elm{
    char *name;
    char *type;
    char *recognizer; /*only when token is terminal*/
    char direction; /*SYNTHESIZED or INHERITED*/
    int arg_count;
    char **arg_type;
    struct list_elm *next, *previous;
    int order;
};

typedef struct list_elm AggListElement;
\end{verbatim}

where
\begin{itemize}
    \item name \texttt{attribute\_name}.
    \item type \texttt{attribute\_type (data\_type)}.
    \item recognizer \texttt{recognizer\_associated\_with\_the\_terminal\_(if\_any)}.
    \item direction \texttt{SYNTHESIZED\_or\_INHERITED\_attribute}.
    \item arg\_count \texttt{reserved\_for\_functions}.
    \item arg\_type \texttt{reserved\_for\_functions}.
    \item next \texttt{next\_element\_in\_the\_list}.
    \item previous \texttt{previous\_element\_in\_the\_list}.
\end{itemize}
At this level, the data member recognizer is always set to NULL. Once the attribute is selected for a given terminal, AGG creates a new copy of the same attribute and adds it to the selected token. If the selected token is of type TERMINAL and has an attribute associated with it, the user can associate a recognizer with the given attribute. The attribute list is used by AGG as a reference for all currently available attributes for the user to select from and to associate with a given token.

The data member recognizer is a string representing the recognizer supplying the terminal with its synthesized attribute (if any), having this element in the attribute template, means that the same recognizer will be applied to all attribute instances associated with a given token.

**A TOKEN’S ATTRIBUTE LIST** Associated with every token is a list of attributes (the member attribute) similar to the attribute list defined earlier (attribute is a subset of the original attributes list), it consists of only those attributes associated with the token. If no attributes are associated with the token the data member attribute is set to NULL. When a new instance is added to the list of token instances, the token’s attribute-list acts as a templet for creating a list of type AggAttribute for the new instance.
The type *AggAttribute* is defined in AGG as follows:

```c
typedef struct attribute AggAttribute;
struct attribute{
  AggListElement *attr;
  char *initial_val;
  char rule_type;
  AggFunction *function; /*NULL when rule.type = INTI_RULE*/
  AggAttribute *next;
  int order;
}; /*attribute structure*/
```

where
- `attr` corresponding attribute in the `token.attribute.list`.
- `initial_val` initial value associated with the attribute (if any).
- `rule_type` one of the predefined constants APPLY_RULE, COPY_RULE, or INTI_RULE
- `function` an instance of function associated with attribute.
- `next` next attribute in the list.

Notice that all instances are linked to their generic type, this ensures that information such as *name*, *direction*, and *type* of attribute is not duplicated in attribute instances.

**FUNCTION LIST** For every function defined by the user or loaded from a library file, AGG creates an element of type *AggListElement* and adds it to the list of functions associated with the program. Function list and attribute list are
both instances of the same data type, however, the components of their elements
are interpreted differently. The following, for example, is how $AggListElement$
components are interpreted when it is a member in a function list:

name function name.
type the type of data returned by the function.
recognizer reserved for attributes.
direction reserved for attributes.
arg_count number of arguments associated with the function.
arg_type array of strings, each of which represents the type of
one argument in the.
next next function in the list.
previous previous function in the list.

FUNCTIONS AGG uses function-list elements as a template for the type
$AggFunction$ associated with some attributes. The type $AggFunction$ is defined
by AGG as follows:

```c
typedef struct{
    AggListElement *func; /*point to a function-list element*/
    AggArgument *args;
    int arg_count;
}AggFunction;
```

where

- `func` points to the template function.
- `args` list of arguments.
- `arg_count` number of currently used arguments.

Notice that all functions are linked to their prototypes, this is because information
such as `function name`, `arg_type`, `type`, and `arg_count` are not duplicated in function
instances.

ARGUMENTS LIST Associated with every function-instance a list of argu-
ments. Arguments are instances of the type $AggArgument$ which is defined in
AGG as follows:

```c
struct arguments{
    AggElement *elm;
    AggAttribute *attribute;
    struct arguments *next, *previous;
};
typedef struct arguments AggArgument;
```

where
elm points to an instance of token.
attribute points to an attribute associated with the token.
next next argument in the list.
previous previous argument in the list.

The links between the arguments, token and attributes are automatically established by AGG while constructing the attribute grammar.

Figure 13 argument list
Section 2 ANALYSIS OF THE INTERNAL REPRESENTATION

In figures (12.2, 12.3, and 12.4), instances to the left of \textit{prd1}, \textit{prd2}, and \textit{prd3} are parts of the instances list associated with the nonterminal \textit{exp}. When carefully examining the structure in figure (12) especially (12.4) we can see that it is a two-dimensional linked list, that is, elements in the list (excluding the head of the list) are members in two lists at the same time (production instances list and token instances list). There are two advantages for having the data structured in this form:

1. \textbf{To simplify the process of translation:} The multiple link provides an efficient direct access to all components relevant to the translation process.

2. \textbf{To improve the efficiency of the dynamic checking process:} By performing the checking only on objects that are affected by the changes. However, we must point out that currently AGG does not support dynamic nested checking but the concept was one of the factors that affected the design of the internal representation. We think that the current structure can support such checking if the appropriate algorithm is developed. The algorithm must provide a method for flagging the objects affected by the changes and a logic to guard agents circularity.

To show how important is the structure for number (2) above, consider the following example:

40
Suppose that the user decided to change the name of the nonterminal \( \text{exp} \) to \( \text{expression} \). Since there are more than one instance of \( \text{exp} \), the change must propagates to all productions that have one or more occurrence of the symbol. To avoid overall restructuring of the grammar, AGG first performs the required changes on each instance and then sends a message to all involved productions by traversing the list of instances associated with the token \( \text{exp} \) figure(11). Upon receiving the message, each production responds by invoking a re-computation routine, which is in this case restructuring the graphical representation.

Figure 14 dynamic checking messages

One disadvantage associated with such data structure (which we experienced ) is the complexity of the management process. Updating such structure (adding and deleting elements) is a complicated task and require a substantial amount of coding and checking (object oriented programming paradigm might be a better
environment for implementing such structure). the same apply to the process of saving and retrieving the structure. However, the advantages we gain from using the structure easily out-weight this single disadvantage.
Chapter 4  TRANSLATION

Translation in AGG is the process of generating a specification for a target system. It is the process of translating the information inside AGG's data structure to a syntactically correct specification. A translator in AGG is a separate module whose sole purpose is to read information from the dynamically-created structure and translate it to a text output in some desired format. Usually translators invoke an overall checking process to determine whether the attribute-grammar or the underlying context-free grammar represented by the structure satisfies all conditions set by the target system.

Section 1 UNDERSTANDING THE STRUCTURE

The main purpose behind designing and building AGG is to provide an external graphical representation and an internal structural representation for attribute grammars, translating such information to a specification for a given system is an external function left to be written by a second party. To write a translator for a given system, the programmer must assume complete knowledge about:

1. The target system.
2. AGG's data structure.
NAVIGATING THROUGH THE STRUCTURE

The multiple links associated with each element in the structure, provides a flexible path for the programmer to navigate through the entire structure, regardless of where the path originated from. Given the address of any token, one can access all of its instances and their productions by a simple traversal to the instances-list associated with the token. Having the instance address, one can access all its attributes, functions associated with the attributes, and arguments associated with each function. Such a flexible yet hard-to-manage structure, provides a direct and easy access to all necessary information required by the translation process, which means not only speeding up the translation, but also minimizing the amount of code required for building a translator. Almost all lists in AGG are doubly-linked, i.e. the program can navigate through the structure in two direction. The previous and the next elements are always directly accessible via the members next and previous except in the case of token instances the two members are defined as left and right to avoid name clashing (because the other two members are used to address the previous and next instances associated with the production).

Figure 15 links associated with each instance

```
previous         next

      instance of
      token

left         right
```
GENERATING A GRAMMAR

A grammar in a BNF-like notation can be generated by a single traversal to AGG’s data structure. To illustrate this, we introduce the following function which generates the context-free grammar associated with the attribute-grammar defined by the structure:

```c
int GenGrammar(AggProgram *pgm)
{
    AggToken *token = pgm->first;
    AggProductionLine *line;
    AggElement *elm;
    while(token){ /*traverse the tokens list*/
        if(token->type == NONTERMINAL){
            printf("%s :\n", token->root->name);
            line = token->first_line;
            while(line){ /*traverse the productions list*/
                printf("\t");
                elm = line->first;
                /*traverse the production instances list*/
                while(elm){
                    printf("%s", elm->name);
                    elm = elm->next;
                } /*while*/
                printf("\n");
                line = line->next;
            } /*while*/
        } /*if*/
        token = token->next;
    } /*while*/
} /*GenGrammar*/
```

The function traverses three lists using three `while` loops. When executed, the function visits all the nonterminals and productions associated with them. Once in the production, the function traverse all instances associated with the production and prints their names. Notice that the token type is tested to determine whether to visit a given token or not. In many cases determining the token type is very important since terminals and nonterminals are both defined to be of the same
type (*AggToken*). When applied on the structure shown in figure(12), the function generates the following grammar:

```
exp :  
  term + exp
  term - exp
  term
  term ::
    factor * term
    factor / term
    factor
    (exp)
    number
```

The same traversal technique can be used to extract information about attributes form other levels in the structure. The order in which the translator visits elements of the structure determines the format of the output. Separators and other control characters are inserted by the translator according to the syntactical requirement of the target system.

**Section 2 CHECKING**

Code generation is a mechanical translation form specially structured attribute-grammar notation to some type of notation accepted by the target system. The correctness of the generated code depends completely on the correctness of the underlying attribute grammar. In all cases, AGG ensures that the generated code is syntactically correct, however, there are very little that can be done to ensure that it is semantically correct. AGG provides *dynamic* and *overall* checking.
DYNAMIC CHECKING

Dynamic checking routines are automatically invoked by AGG whenever the action of the user reflect changes on the internal structure of the program. The checking process usually covers parts of the structure that are most possibly effected by the changes; however, there are cases when the process expands to cover the entire structure. AGG provides two types of dynamic checking:

1. **Left-recursion** checking.
2. **Type** checking.

**LEFT-RECURSION**  Consider the following simple production:

\[ \text{exp} ::= \text{exp} \circ \text{term} \]

In top-down parsing, expanding the nonterminal \textit{exp} will leads to the same non-terminal being expanded again and again without ever reaching to a terminating condition. Productions similar to the one above are referred to as **left-recursive**. When the target system (the system for which code to be generated) adopts a top-down parsing strategy, left-recursion checking must be conducted before the translation process, otherwise, the generated code, although syntactically correct, will leads to unrecoverable run-time errors. Detection of direct left recursion (when the first token in the right-hand side is the same as the one in the left-hand side) is quite simple. However, left recursion is not always direct, there are cases when left-recursion appears only after multiple expansion processes:
(1) exp ::= term + exp
(2) term ::= factor * term
(3) factor ::= exp

Notice that all the productions above are indirectly left recursive. To check
whether production (2) is left recursive we do the following:

if the first token in the right-hand side is a terminal
report failure
fi
else
if the first token in the right-hand side = term
report success
fi
else
expand factor and start from beginning
esle
esle

Following the above algorithm left recursion on production (2) is detected as
follows:

term ::= factor * exp || expand factor
term ::= exp * term || expand exp
term ::= term + exp * term || direcule-recursion

The problem with detecting this type of left recursion is that the checking process
itself can be trapped in the non-terminated expanding process. Consider the
following grammar:

(1) exp ::= term + exp
(2) term ::= factor * term
(3) factor ::= term

Suppose we want to check whether the exp is left recursive using the same
algorithm above:

exp ::= term + exp expand term (2)
exp ::= factor * term + exp expand factor (3)

48
\[ \text{exp : } \text{term} \cdot \text{term} + \text{exp} \]

Notice that the checking process itself got trapped in the left recursion while expanding \text{term}. This problem can be avoided by controlling the number of recursive calls with a counter. This approach is adopted in AGG and implemented in the following function:

```c
AggProductionLine *AggIsLeftRecursive(prd, line, count)
AggToken *prd;
AggProductionLine *line;
int count;
{
    AggProductionLine *tmp;
    AggElement *elm;
    /*count the depth of the recursive call*/
    if(count == 0)
        return(NULL);
    while(line){
        /*start with the first line*/
        elm = line->first;
        /*skip elements marked for deletion*/
        while(elm && elm->status)
            elm = elm->next;
        /*check only nonterminals*/
        if(elm && elm->type == NONTERMINAL){
            /*if token is the same as parent*/
            if(elm->parent_token == prd)
                /*report left recursion*/
                return(line);
            if((tmp =
                AggIsLeftRecursive(prd,
                elm->parent_token->first_line,
                count - 1)))
                return(tmp);
        }
        line = line->next; /*check the next line*/
    }
    return(NULL); /*no left recursion detected*/
    /*AggIsLeftRecursive*/
}
```

The third argument \text{count} represents the maximum number of recursive calls allowed including the current call, this argument is decremented by one every
time the function calls itself recursively. When the function *AggIsLeftRecursive* invoked by the user, *count* must be greater or equal to the number of nonterminals in the given grammar. *AggIsLeftRecursive* is recursively called whenever the first token in the right-hand side is a nonterminal and not equal to the given left hand side nonterminal *prd*, i.e., a recursive call for every nonterminal being expanded. When the number of recursive calls accede the number of nonterminals in the grammar, this mean that one or more of the nonterminals has been expanded more than one time, an indicator for the existence of left-recursion.

**TYPE CHECKING**  
AGG associates types with attributes, functions, and arguments. When the user define new attribute, AGG prompts the user for a name, a direction (synthesized or inherited), and a data type. The type-checking process is automatically invoked whenever the user attempts to link two constructs associated with data type. The result of the checking process will determine whether to allow the current action or not. The type-checking function decomposes the given types to their atomic components. For example, the type (*int, real, [char]*) is decomposed to the following set of atomic components:

```
"(" "int" "," "real" "," ")" "(" "char" ")"
```

Corresponding components are then compared to each other, failure will be reported at the first mismatch. The type "*" in the left hand side can match any atomic type in the right hand side, however, "*" in the right hand side can match
only "*" on the left hand side. An atomic-type, as recognized by AGG, is an identifier that consists entirely of one or more of the following characters \{(A-z), (1-9), '_}\}. Some characters are defined to have special meaning. The following is a list of all the special characters and their interpretation:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>tuple</td>
</tr>
<tr>
<td>[]</td>
<td>list</td>
</tr>
<tr>
<td>*</td>
<td>any atomic type</td>
</tr>
<tr>
<td>-&gt;</td>
<td>function</td>
</tr>
<tr>
<td>comma</td>
<td>separator for tuple’s and function’s elements</td>
</tr>
<tr>
<td>space</td>
<td>separator</td>
</tr>
</tbody>
</table>

AGG imposes no restrictions on the type name, however, programmers can have their own checking routines to determine whether or not a given data type is illegal. Type checking routines (if any) must be added to the overall checking routines associated with the translator. The following is a complete listing of AGG’s type checking functions:

```c
int AggCheckType(char *type1, char *type2)
{
    char left[60];
    char right[60];
    int match;
    char *l_ptr, *r_ptr;
    if(!strcmp(type1, type2))
        return(1);
    l_ptr = type1;
    r_ptr = type2;
    while((match =
        AggCompareType(AggGetType(left, &l_ptr),
                        AggGetType(right, &r_ptr)))):
        if(match)
            return(1);
```
return(0);
 /*!<AggCheckType*/

/*!compare two atomic types, return zero if match*/
int AggCompareType(char *left, char *right)
{
    if(left == NULL || right == NULL)
        return(0);
    if(strcmp(left, right))
        return(1);
    if(!strcmp("\[\]", right[0]) && strcmp(left, "*"))
        return(1);
    return(0);
 /*!<AggCompareType*/

The following table, show some possible types and the result of comparing them to each another:

<table>
<thead>
<tr>
<th>TYPE 1</th>
<th>TYPE 2</th>
<th>TYPE MATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>TRUE</td>
</tr>
<tr>
<td>(int, real)</td>
<td>(int, int)</td>
<td>FALSE</td>
</tr>
<tr>
<td>(int, (real, *), int)</td>
<td>(int, (real, int), int)</td>
<td>TRUE</td>
</tr>
<tr>
<td>*</td>
<td>(int, real)</td>
<td>FALSE</td>
</tr>
<tr>
<td>[char]</td>
<td>[*]</td>
<td>FALSE</td>
</tr>
<tr>
<td>*-&gt;int</td>
<td>real-&gt;int</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Type-checking can be disabled from option menu.

OVERALL CHECKING

Before starting the translation process, the translator usually invokes an overall check to insure that all the target-system requirements are met by the currently-structured attribute grammar. The overall checking is a process of traversing the whole structure and performing some system dependent (target system) tests. This
test must be designed and implemented together with the translator. In most cases, the check involves some key points about the target system. For example, the availability of the start-symbol is crucial to YACC translator, therefore, it must be checked by the overall checking process associated with YACC’s translator. AGG performs its own overall check, by applying the dynamic check on all the elements of the current structure.
Chapter 5 BUILDING A TRANSLATOR

As mentioned earlier, a programmer must completely understand the AGG internal representation of attribute grammars and the syntax and semantics of the target system, before attempting to build any translator. In this chapter we overview W/AGE as an attribute-grammar system, and using our knowledge of AGG's internal structure (see chapter 3), we will carefully examen some of the important stages in designing and building a translator for W/AGE.

Section 1 W/AGE

The Windsor Attribute Grammar Programming Environment was developed by Frost (1990) as a response to the need for specialized language processors for projects in the area of database management and natural-language design. The idea behind W/AGE is to allow the construction of executable specifications of a given program, that is, constructing language processors using W/AGE is merely a process of specifying the syntax and semantics of the required language, another step towards what are referred to as very high level programming languages. The core of W/AGE consists of a set of functions built as an extension to the standard environment of the pure functional programming language MIRANDA. However, the current W/AGE implementation is more of a specification of W/AGE than it is an application, that is, changing the underlying functional language will have very
little or no effect on W/AGE style. MIRANDA was chosen for many reasons, the most important one is the support of lazy-evaluation, a fundamental concept in the W/AGE system. As mentioned earlier, W/AGE consists of several functions that can be categorized according to their task, which ranges from scanning the input tokenize, constructing interpreters (literal, interpreted, uninterpreted), combining interpreters (Sorelse, Sexcl_orelse, structure) and applying interpreters (apply_interpreter, apply_recogniser). As an attribute grammar environment W/AGE provides mechanisms by means of which attributes can be built from the basic construct terminal, which is defined in W/AGE over a sufficient range of types, and other types supported by the host programming language. The following is the type terminal as it is defined in W/AGE, where :: declares the left-hand side to be of the type given on the right-hand side:

```
terminal ::= INT_TERM [char]
           | IDENTIFIER_TERM [char]
           | RESERVED_WORD_TERM [char]
           | ANY_TERM [char]
           | REAL_TERM [char]
           | SPECIAL_SYMBOL_TERM [char]
           | UNCATEGORYISED_TERM [char]
```

Words written in capital letters are type constructors. notice that all types are constructed from the type [char], a basic type supported in almost every programming language. Attributes are constructed in similar fashion, however, there are two minor differences:

1. Attributes are named and defined by the programmer.
2. The building components of attributes are the types terminal and any other simple or structured type supported by the host programming language.

Allowing direct access to the host programming language (which in this case is functional) provides W/APE with a very general notion of value (which functional programming languages are known to have Johnsson[7]) that is very well suited for attributes, which can be anything between simple character to a sophisticated function. The following code fragment defines some attributes:

```
attribute ::= LITERAL_VAL terminal
          | VAL num
          | OP num -> num -> num
          | PAIR (num [char])
```

The first attribute is of type terminal, the second is of type num, the third is a function and the forth is a tuple. The only restriction imposed on attribute types is whether or not the type is supported by the host programming language.

Interpreters in W/APE are defined to have the following type:

```
interpreter = [(attribute, [terminal])] ->
               [(attribute, [terminal])]
```

Notice that the type of the output is the same as that of the input, this is to allow a subsequent application of other interpreters on the output (the result) of a given interpreter. The input is a list of pairs each of which consist of two components, a list of attributes and a list of terminals. Terminals are interpreted using the attributes as a context. Pairs in the output are of the same type as those in the
input; however, their components reflect the task accomplished by the interpreter. The first component (attribute) is a subset of the first component of the input and the successfully interpreted part (initial part) of the second component (terminal).

The second component (terminals) is the part that is yet to be interpreted. A pair in the input with more than one possible interpretation corresponds to more than one pair in the output, each of which represent one interpretation. Interpreters are built using specialized functions defined in W/AGE, (literal, interpreted, uninterpreted).

These functions are defined to have the following types:

\[
\begin{align*}
\text{literal} & \quad : \text{terminal_constructure} \rightarrow \text{interpreter} \\
\text{uninterpreted} & \quad : \text{terminal} \rightarrow \text{interpreter} \\
\text{interpreted} & \quad : \{\text{terminal, [attribute]}\} \rightarrow \text{interpreter}
\end{align*}
\]

Interpreters built using the functions above are referred to as basic interpreters.

Building basic interpreters in W/AGE is a simple process, which starts by specifying a name, and then using one of the three building functions with a suitable argument depending on the type of interpretation required. The following code fragment represents a process of building three basic interpreters:

\[
\begin{align*}
\text{real} & \quad = \text{literal} \quad \text{REAL_TERM} \quad (1) \\
\text{plus} & \quad = \text{uninterpreted} \quad (\text{SPECIAL_SYMBOL_TERM} \ "\+"") \quad (2) \\
\text{one} & \quad = \text{interpreted} \quad (\text{RESERVED_WORD_TERM} \ "\text{one}\" \ [\text{VAL} \ 1]) \quad (3)
\end{align*}
\]

Interpreter \textit{plus} from equation (2) is a language processor which recognizes the symbol "+" but does not compute any attribute for it. The use of the terminal type \textit{SPECIAL_SYMBOL_TERM} requires a declaration of a \textit{special_symbols} (characters) list, usually placed in the beginning of W/AGE programs after the
declaration attribute types. A similar list of reserved_words (strings) is usually
declared after the special_symbols list. If no reserved_words or special_symbols
are used in the program, the corresponding list is defined to be empty.

In most cases basic interpreters are not sufficient for building complete
language processors. Three important functions (Sorelse, Sexcl_orelse, structure)
introduced by W/A GE for the purpose of combining interpreters to build a new
one capable of recognizing and interpreting more complicated input. The three
functions are referred to as interpreter combinators, and defined to have the
following types:

Sorelse :: interpreter -> interpreter -> interpreter
Sexcl_orelse :: interpreter -> interpreter -> interpreter
structure :: list_of_tagged_interpreters ->
            list_of_attribute_rules -> interpreter

where
list_of_tagged_interpreters == [{tag, interpreter}]
list_of_attribute_rules == [{rule_num, att_id,
                            att_function, [att_id]}]

where
att_id== {(tag, att_direction), att_type}
att_function == [attribute] * attribute
att_type== [char]
att_direction::= $u | $d(UP or DOWN)

Functions prefixed with symbol ‘$’ are used as infix operators. att_direction is
used to specify whether an attribute is inherited or synthesized (the latest version of
W/A GE requires no such definition. $u and $d may be replaced by the function $of
). Sorelse Sexcl_orelse are very simple to use, however, structure requires some
familiarity with W/A GE and MIRANDA notation, the first helps in choosing the
right rule, and the second in building functions for attribute computation. Three
types of attribute computation rules are defined in \textit{W/AGE}:

1. \textit{a\_rule}: apply rule.
2. \textit{c\_rule}: copy rule.
3. \textit{i\_rule}: initialization rule.

There are no restrictions on the size and complexity of the attribute function, however the type of the value returned by the function must matches the type of the attribute for which it is assigned to. When \textit{copy} and \textit{apply} rules are used, attribute computation required values of other attributes initialized or computed by other equations in the same program. The order of attribute evaluation is resolved at execution time by the \textit{lazy-evaluation} mechanism built into the host functional programming language, as a result circularity will no more be viewed as an obstacle, for which absence must be established before starting the evaluation process.

\textbf{EXAMPLE}

The following is the well-known mathematical-expression attribute grammar:

\begin{align*}
\text{expr1} & \rightarrow \text{term} \text{ plus } \text{expr2} \\
\quad \text{expr1} \uparrow \text{VAL} & := \text{term} \uparrow \text{VAL} + \text{expr2} \uparrow \text{VAL} \\
\text{expr1} & \rightarrow \text{term} \text{ minus } \text{expr2} \\
\quad \text{expr1} \uparrow \text{VAL} & := \text{term} \uparrow \text{VAL} - \text{expr2} \uparrow \text{VAL} \\
\text{expr1} & \rightarrow \text{term}
\end{align*}
expr|VAL := term|VAL

term1 := factor times term2

  term1|VAL := factor|VAL * term2|VAL

term1 := factor divide term2

  term1|VAL := factor|VAL / term2|VAL

term1 := factor

  term1|VAL := factor|VAL

factor := opbr expr cbr

  factor|VAL := expr|VAL

factor := number

  factor|VAL := number|VAL

number := real_num

  number := literal_val(real_num)

number := int_num

  number|VAL := literal_val(int_num)


The corresponding W/AGE specification will be as follows:

%insert<local/header_for_WAGE_VERSION_2_RELEASE_0.m>
%insert "functions.m"
||define required attributes
attribute ::= LITERAL_VAL terminal
  | VALUE num

||define the set of reserved words used by grammar
reserved_words = {}
declare the set of special symbols used by the grammar
special_symbols = ['+', '-', '*', '/', '

|basic interpreters
opbr = uninterpreted (SPECIAL_SYMBOL_TERM "(")
cibr = uninterpreted (SPECIAL_SYMBOL_TERM ")")
minus = uninterpreted (SPECIAL_SYMBOL_TERM "-"")
plus = uninterpreted (SPECIAL_SYMBOL_TERM "+")
times = uninterpreted (SPECIAL_SYMBOL_TERM "**")
divide = uninterpreted (SPECIAL_SYMBOL_TERM "/")
real_num = literal REAL_TERM
int_num = literal INT_TERM

|interpreters built by combining basic interpreter
expr = structure (s1 term ++ s2 plus ++ s3 expr)
       [a_rule 1.1 (VALUE $u 1hs) EQ add [VALUE $u s1, VALUE $u s3]]
       $orelse
       structure (s1 term ++ s2 minus ++ s3 expr)
       [a_rule 1.2 (VALUE $u 1hs) EQ sub [VALUE $u s1, VALUE $u s3]]
       $orelse
       structure (s1 term)
       [c_rule 1.3 (VALUE $u 1hs) EQ (VALUE $u s1)]

term = structure (s1 factor ++ s2 times ++ s3 term)
       [a_rule 2.1 (VALUE $u 1hs) EQ multiply [VALUE $u s1,VALUE $u s3]]
       $orelse
       structure (s1 factor ++ s2 divide ++ s3 term)
       [a_rule 2.2 (VALUE $u 1hs)EQ divide_by [VALUE $u s1,VALUE $u s3]]
       $orelse
       structure (s1 factor)
       [c_rule 2.3 (VALUE $u 1hs) EQ (VALUE $u s1)]

factor = structure (s1 opbr ++ s2 expr ++ s3 cibr)
       [c_rule 3.1 (VALUE $u 1hs) EQ (VALUE $u s2)]
       $orelse
       structure (s1 number)
       [c_rule 3.2 (VALUE $u 1hs) EQ (VALUE $u s1)]

number = structure (s1 real_num)
       [a_rule 4.1 (VALUE $u 1hs) EQ atof [LITERAL_VAL $u s1]]
       $orelse
       structure (s1 int_num)
       [a_rule 4.2(VALUE $u 1hs) EQ atoi [LITERAL_VAL $u s1]]

The file functions.m, which defines the required MIRANDA function, must be in the same working directory. The following is the listing of the file functions.m:

|attribute computation functions
add [VALUE x, VALUE y] = VALUE (x+y)
sub [VALUE x, VALUE y] = VALUE (x-y)
multiply [VALUE x, VALUE y] = VALUE (x*y)
\[
\begin{align*}
divide\_by \ [\text{VALUE } x, \text{VALUE } y] &= \text{VALUE } (x/y) \\
\text{atof} \ [\text{LITERAL\_VAL (REAL\_TERM } x)] &= \text{VALUE (numval } x) \\
\text{atoi} \ [\text{LITERAL\_VAL (INT\_TERM } x)] &= \text{VALUE (numval } x)
\end{align*}
\]

The main task of the translator is to generate code similar W/AGE specification above, given the internal representation of AGG.

**Section 2 THE TRANSLATOR**

Translation is a mapping from the source (attribute grammar) to the target (code for attribute system). Attribute systems usually view attribute grammars from their own perspective, which is usually determined by many factors, such as their host languages, the parsing algorithm, the implementation language, ... etc. The main functionality of the translator is to transform the attribute grammar components into a new form accepted by the target system.

**TERMINALS**

Terminals are mapped to one of the following three functions:

1. literal.
2. uninterpreted.
3. interpreted.

The function *WageGenTerminal* checks the attribute list associated with the terminal. If the list is not empty (*token->attribute->list != NULL*) the third mapping is selected. Otherwise (when the list is empty), the function checks the pattern
associated with the token (token->pattern) by calling the function WageGetPatternType which returns one of the following predefined constants:

1. **SYMBOL**: if the pattern is a single character, the mapping in this case is:
   
   `uninterpreted (SPECIAL_SYMBOL_TERM, "pattern")`.

2. **LITERAL**: if the pattern is a member in the predefined RecognizersList, the mapping in this case is:
   
   `literal "pattern"`.

3. **WORD**: otherwise, the mapping is
   
   `uninterpreted (RESERVED_WORD_TERM, "pattern")`.

The following is a complete listing of the function WageGetPatternType

```c
char WageGetPatternType(char *pattern)
{
    if(!pattern)
        return(0);
    if((strlen(pattern) == 1) && isalpha(pattern[0]))
        return(SYMBOL);
    if(AggReservedWord(RecognizersList, pattern) >= 0)
        return(LITERAL);
    return(WORD);
} /*WageWordOrSymbol*/
```

Figure 16 mapping of terminal

- interpreted **associated with attribute**
- literal **pattern is a recognizer**
- uninterpreted **otherwise**

63
Once the proper mapping is established, the rest is a simple output formatting.

The following is a complete listing of the function *WageGenTerminal*

```c
int WageGenTerminal(FILE *handle, AgoToken *token,
                      char *words,
                      char *symbols)
{
    AgoListElement *att;
    char *tmp;
    if(!token)
        return(0);
    att = token->attribute->list;
    tmp = token->root->name; /*token_name*/
    fprintf(handle, "\%s = ", token->root->name);
    if(!att)
        switch(WageGetPatternType(token->pattern)){
            case WORD:
                fprintf(handle,
                        "uninterpreted(RESERVED_WORD_TERM \"%s\")\n",
                        token->pattern);
                if(words[0] == '\0';
                    fprintf(handle, "\"%s\", \"%s\", token->pattern);
                else
                    sprintf(PgmInfo.MsgBuff, ",\"%s\", token->pattern);
                strcat(words, PgmInfo.MsgBuff);
                "/\";break;
                case SYMBOL:
                    fprintf(handle,
                            "uninterpreted(SPECIAL_SYMBOL_TERM \"%s\")\n",
                            token->pattern);
                    if(symbols[0] == '\0';
                        fprintf(handle, "\"%s\", token->pattern);
                    else
                        fprintf(PgmInfo.MsgBuff, ",\"%s\", token->pattern);
                    strcat(symbols, PgmInfo.MsgBuff);
                    }else*/
                break;
                case LITERAL:
                    fprintf(handle, "literal \%s\n", tmp);
                    break;
                }/*switch*/
            }/*if*/
        else
        if(words[0] == '\0';
            fprintf(handle, "\"%s\", token->pattern);
        else
            sprintf(PgmInfo.MsgBuff, ",\"%s\", token->pattern);
        strcat(words, PgmInfo.MsgBuff);
        }/*else*/
    fprintf(handle, "\tinterpreted (RESERVED_WORD_TERM \"%s\", \"%s\")\n",
```
Notice that the name of the token can be retrieved via the data member `root` (a structure of type `AggElement`) which contains the address of the name in its data member `name` (`token->root->name`). The arguments (words and symbols) are addresses of the `reserved_words` and `special_symbols` lists. When the mapping is determined to be of type `uninterpreted symbol`, the function attaches the current pattern to the `special_symbols` list, otherwise, if the mapping is of type `uninterpreted word` the current pattern is attached to `reserved_words` list.

**NONTERMINALS**

Nonterminals in AGG, are tokens with at least one `production_line`. The function `WageGenNonterminal` performs mapping of the form:

```
AggProductionLine -> WAGE_structure
```

When more than one `production_line` is associated with a nonterminal, the function inserts the combinator `$orelse$` between the corresponding `WAGE_structures`. The combinator `Sexcl_orelse` will not be generated.

The main functionality of `WageGenNonterminal` is to traverse the list of `production_lines` associated with the nonterminal, and calls the function `WageGenStruct` for every element in the list. The following is a complete listing of the function `WageGenNonterminal`:
int WageGenNonterminal(FILE *handle, AggToken *token)
{
    AggProductionLine *line = token->first_line;
    if(!token)
        return(0);
    printf(handle, "%s\t\n", token->root->name);
    while(line)
    {
        WageGenStruct(handle, line);
        if(line->next)
            printf(handle, "\t\t\t\t\n");
        line = line->next;
    } /*while*/
} /*WageGenNonterminal*/

The address of the list of production_lines is stored in the data member first_line (token->first_line).

INTERNAL REPRESENTATION OF PRODUCTIONS  A Production, as viewed by AGG, is a list of ordered token instances. The first instance in the list represents the left-hand side of the production. Elements of the list are ordered from left to right, the first element (LHS) has the order (-1). The function Wage-GenStruct traverse the list of instances twice. In the first traversal, it perform the following map:

Figure 17 mapping production to WAGE-structure

AggProductionLine

LHS      elm->order      elm->name

  ▼ (-1) expr  ▼ (0) term  ▼ (1) plus  ▼ (2) expr

expr = structure (s1 term ++ s2 plus ++ s3 expr) [rules]

The second components of WAGE-structure ([rules]) is generated by a sec-
ond traversal to the list. The following is a complete listing of the function

\textit{WageGenStruct}:

\begin{verbatim}
int WageGenStruct(FILE *handle, AggProductionLine *line)
{
    int count = 1;
    AggElement *elm = line->first;
    fprintf(handle, "\t\nstructure(");
    while(elm){
        if(!elm->previous)
            fprintf(handle, "s%d %s ", count, elm->name);
        else
            fprintf(handle, "s%d %s ", count, elm->name);
        elm = elm->next;
        count++;
    }/*while*/
    fprintf(handle, "}
\n");
    elm = line->first;
    fprintf(handle, "\t\n\t\nWageGenRules(handle, line->lhs, &count);
    while(elm){
        WageGenRules(handle, elm, &count);
        elm = elm->next;
    }/*while*/
    fprintf(handle, "}\n\n");
}/*WageGenStruct*/
\end{verbatim}

In the second traversal, the function \textit{WageGenRules} is called for every element in the list to generate the set of rules associated with each element. The argument \textit{count} of \textit{WageGenRules} keeps track of the current rule number.

\textbf{PRODUCTION RULES} Instances of tokens are usually associated with a list of attributes. The value of each attribute is determined by either a \textit{rule} associated with the attribute, or by a \textit{recognizer} (only when the token is a terminal). AGG recognizes three types of rules (\textit{APPLY_RULE}, \textit{COPY_RULE}, \textit{INTI_RULE}). The function \textit{WageGenRules} traverse the list of attributes associated with a token, and calls the function \textit{WageGenRule} for every element in the attribute list:
Given an attribute, the function `WageGenRule` determines the type of the rule associated with it and generates code accordingly. The type of the rule is determined by checking the data member `rule_type`, which must be one of the predefined constants (`APPLY_RULE`, `COPY_RULE`, `INTI_RULE`).

The following is a complete listing of the function `WageGenRule`:

```c
int WageGenRule(FILE *handle, AggElement *elm, int *order)
{
    AggAttribute *att = elm->attribute;
    while(att)
    {
        if(WageGenRule(handle, elm, att, *order))
            (*order)++;
        att = att->next;
    }/*while*/
    }/*WageGenRules*/
```

```c
int WageGenRule(FILE *handle, AggElement *elm, int *order)
{
    switch(att->rule_type){
    case APPLY_RULE:
        if(order)
            fprintf(handle, "\n\n\n");
        fprintf(handle, "a\_rule \%d,\%d (%s $%c", Count,
            SubCount,
            att->attr->name,
            att->attr->direction);
        SubCount++;
        WagePrnElmOrder(handle, elm);
    case COPY_RULE:
        break;
    case INTI_RULE:
        break;
    default:
        break;
    }
    }/*WageGenRules*/
```
When the rule_type is APPLY_RULE, WageGenRule calls the function WageGenFunction to generate the proper function, which pointed to by the member data function.

The function WagePrnElmOrder prints the order of the element in the instances list, or lhs if the instance is the left-hand side of the production. The order of the elements is determined by adding one to the data member element->order.

int WagePrnElmOrder(FILE *handle, AggElement *elm)
{
    if (!elm)
        return (0);
    if (elm->order >= 0)
        fprintf(handle, "%d

69
FUNCTION  As shown in chapter 3, the type **AggFunction** contains the data
member *func*, which points to the function associated with the given attribute (used
to retrieve the function name *fun-* > *func-* > *name*), array of type **AggArgument** (used
to generate the list of arguments associated with the function), and an integer
representing the number of arguments. The function **WageGenFunction** maps
the information in **AggFunction** structure to a WAGE-style function call. Each
argument in the array *fun-* > *args* points to an instance *fun-* > *args[i]-* > *elm* and an
attribute associated with instance *fun-* > *args[i]-* > *attribute*. All the information
necessary to generate WAGE-type argument can be retrieved through these two
pointers.

Figure 19 mapping of function to WAGE-function

A **COPY_RULE** is a function with one argument that is automatically created
by AGG. \textit{WageGenFunction} checks for the rule type and translates to proper notation required by W/AGE. The following is a complete listing of the function

\textbf{WageGenFunction}:

```
int WageGenFunction(FILE *handle, AggFunction *fun, char rule)
{
    int i;
    if(rule == COPY_RULE)
        fprintf(handle, "(");
    else
        fprintf(handle, ".%s\", fun->func->name);
    for(i = 0; i < fun->arg_count; i++)
    {
        if(i > 0)
            if(fun->args[i]->attribute && fun->args[i]->
                attribute->attr->recognizer)
                fprintf(handle, ", LITERAL\_VAL\"");
        else if(fun->args[i]->attribute)
            fprintf(handle, ", %s\", fun->args[i]->attribute->attr->name);
        else
            fprintf(handle, ", %s\", fun->args[i]->name);
    }
    if(fun->args[i]->attribute)
        fprintf(handle, ", $\%s\", fun->args[i]->
            attribute->attr->direction);
    WagePrnElmOrder(handle, fun->args[i]->elm);
    for( ;
        if(rule == COPY_RULE)
            fprintf(handle, ")\"");
        else
            fprintf(handle, ")\"");
    }*/WageGenFunction*/
```
Chapter 6 CONCLUSION

Our work with AGG shows that building a translator that maps the contents of the internal structure to a syntactically correct code for a target attribute grammar system is relatively simple. By providing the system with structural-editing capability, a set of dynamic checking processes and an easy-to-use graphical user interface, we have freed the user from the burden of memorizing and writing unnecessary syntactical details, a consequence of this, the possibility of writing syntactically incorrect code is significantly reduced. The multiple-translator support can help in simplifying the process of porting correct programs to different target systems.
APPENDIX A: USER MANUAL

WHAT IS AGG

The Attribute Grammars Generator is a visual editing tool in which graphical objects can be dynamically created and linked to form an attribute grammar. The program provides facilities for creating and linking three types of primitive objects Terminals, Nonterminals and Attributes. The program also provides several dynamic checking operations to inform the user about the consequences of the currently invoked action. The constructed attribute grammar can then be processed by a translator to generate code for a previously selected tool such as W/AGE (section 4.1).

AGG BASICS

The command line for invoking AGG is:

agg [filename]
where
filename is an AGG file (a file with .agg extension).

When no filename is supplied, AGG starts with an empty file. Figure (20) shows the main components in a typical AGG editing cession.
1. **Menu area.**

2. A **nonterminal**: Each nonterminal in AGG is associated with a *push-button* labelled with its name. The initial nonterminal can be recognized by its distinct color.

3. **Nonterminals area**: All nonterminals are grouped in this area, when the number of nonterminals accede the maximum capacity of this area, a scroll-bar automatically appears to allow for a partial view, similarly when the width of a button accede the width of the area.

4. **Terminals area**: The same as Nonterminals area except that push-buttons in this area correspond to terminals.
5. **A terminal**: Represents a terminal in the grammar.

6. **Message area**: Displays the name of the currently selected production, when no production is selected, a welcome message is displayed. Accelerators description messages are also displayed on this area whenever the mouse pointer passes over one of them.

7. **Drawing area**: A drawing canvas used to build, edit, and view grammars components. The size of the drawing window can exceed the exposed area.

8. **Productions**: A graphical representation for a set of productions that shares the same left-hand side symbol.

9. **Drawing area scroll-bars**: Used to expose hidden parts of the drawing area.

10. **Control area**: Frequently used commands such as *open file, save file* can be invoked from the control area using accelerators. An accelerator is a push button labeled with an icon representing its action, a brief message will be displayed on the user message area whenever the mouse pointer passed over one of the accelerators, the message briefly describes the functionality of the accelerator.
Delet token (drop site).
New Program.
Edit functions for selected production.
Open new file.
Save file.
Insert (orelse) to selected production.
Create new terminal
Create new nonterminal
Hide selected production
Edit production (drop site).

AGG accelerators

A typical AGG cession starts with an empty file, however, you can load and edit existing files. While editing the program, AGG invokes dynamic checking process to insure the syntactic correctness of the program. You can disable any of these process from Setup menu:
Once the attribute-grammar is fully constructed, you can invoke a translator to generate code for pre-selected systems. Currently, AGG generates code for a syntax-tree generator and W/AGE attribute system.

**WORKING ON PROGRAMS** The final result of a successful AGG session is a syntactically correct code for a given attribute system. You can always save finished or unfinished files and retrieve them in future sessions.

When building an attribute grammar, you can start with an empty file or continue working on previously saved file. AGG provides tools for constructing new primitive objects and the means to link, modify, and restructure them.
OPEN NEW FILE  To edit or continue working on previously created file, you must first load the file in AGG's structured memory. You can load files in two ways:

1. Supply the file name as an argument when starting AGG.
2. Use the file menu.

The loading process is a complicated translation from a binary file to an internal and graphical structure recognized by AGG. The process is automatically invoked when AGG recognizes the extension (must be .agg) and the format of the file. Figure (23) shows AGG's file menu and the corresponding set of accelerators:

![Figure 23 file menu]

Accelerators provide a fast way of invoking frequently used commands see figure(21). To open new file, select OpenFile from File menu or clicking on the
appropriate accelerator, you will see the following dialog:

Figure 24 file dialog

Select any file with .agg extension and click on OK button. If you issue the load command while working on another file, AGG will inform you to remove the current file before loading the new one. You can do that by selecting New File from File menu or by clicking on the appropriate accelerator.

**GENERATE CODE** Once the grammar or the attribute-grammar is completely structured in AGG's memory, you can invoke a selected translator from the file
menu:

Figure 25 code generation menu

<table>
<thead>
<tr>
<th>File</th>
<th>Edit</th>
<th>Option</th>
<th>Library</th>
<th>Graph</th>
<th>Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Code</td>
<td>Grammar</td>
<td>WAGE</td>
<td>generate grammar</td>
<td>generate code for WAGE</td>
<td></td>
</tr>
<tr>
<td>Save File</td>
<td>WAGE</td>
<td>WAGE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open File</td>
<td>WAGE</td>
<td>WAGE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td>WAGE</td>
<td>WAGE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before invoking the selected translator, AGG performs an overall check on the constructed attribute grammar to determine whether or not it satisfies all the target system requirements. For example, AGG will not invoke the W/AGE translator if left recursion is detected in the grammar. Missing arguments form semantic rules which are reported as messages in the shell from which you launched AGG.

Figure 26 overall check messages

Each message reports a missing argument from a given function in a given production. For example, the first message in figure(26) informs you that, an attribute of type VAL is missing from the function add associated with the instance expr of production expr. You can also invoke the overall checking by selecting Check Program from the Option menu.
EXIT AND SAVE PROGRAM  To terminate an AGG session, select Exit from
File menu. AGG will prompt you to whether or not to save the current program.
You can also save program by selecting Save File from File menu or by clicking on
the appropriate accelerator, see figure(23). All AGG files must have the extension
.agg. If you supply any thing other than .agg., AGG will adds the proper extension
and save the file under the newly generated name. For example "file1" will be
changed to "file1.agg" and "file2.abc" will be changed to "file2.abc.agg"

BUILDING A PROGRAM

As far as AGG is concerned, a complete program is a complete attribute
grammar, i.e. an attribute grammar that has a rule (a_rule, c_rule or i_rule)
associated with every synthesized attribute in the left-hand side and inherited
attribute in the right-hand side. To build an attribute grammar, first create the
basic components, such as tokens, attributes and functions then use the editing
tools provided by AGG to link these components. The easiest way to build
an attribute-grammar is to build the underlying context-free grammar and then
use the attribute editor to link the appropriate functions to attribute instances.
Once you have defined all the basic objects, generating a program is only few
mouse-clicks a way.

THE BASIC COMPONENTS  Tokens (terminals and nonterminal) are the
building blocks for any grammar. AGG provides facilities for defining, building
and editing tokens. Terminals and nonterminals are separated into two groups. Terminals are allocated in the right-hand side of the drawing area while nonterminals in the left-hand side. Figure(27) shows the New menu and the corresponding accelerators.

![Figure 27 New token menu](image)

**TERMINALS** Terminals are the atomic components in any grammar, they can be defined as *recognizers for language strings*. You can create a new terminal by selecting Terminal from New sub-menu in Option menu, or by simply clicking on the appropriate accelerator. AGG will prompt you with the New Terminal dialog shown in figure(28). The dialog has two entries, one for the terminal name and the second is for the pattern to be recognized by the terminal.
To associate attributes with newly created terminals, click on the button labeled Attribute. A dialog with a list of all currently defined attributes will appear and prompt you for a selection. You can select any attribute or type your own (when the program starts, the list is always empty).
Select an attribute and click on Apply button, AGG adds it to the newly created terminal as a synthesized attribute. If you selected an attribute that is not a member in the list (by typing its name in the selection area), you will see the following dialog.
If you select Yes, AGG will prompts you for the data type of the new attribute. After entering the data type, which can be primitive or structured (see section 4.2), select either SYNTHESIZED or INHERITED to associate the attribute with the token.

Notice that you can only associate a synthesized attribute if the token is a terminal. Once the attribute is selected, all dialogs except the one shown in figure(28), will
disappear from the screen. Click on OK to create a push-button representing
the token in the terminals area.

**NONTERMINALS** Creating nonterminals is very much the same as creating
terminals, except that nonterminals have no pattern associated with them. The
new-nonterminal dialog, as shown in figure(32), prompts you only for the name
of the nonterminal.

![Figure 32 new-nonterminal dialog](image)

When you create a nonterminals, AGG adds a new push-button to the nonterminals
area and draw a square with a dashed line borders in the upper-left corner of the
drawing area representing the common left-hand side of a set of productions. By
following the same procedures above, you can create as many tokens as you want. If the number of tokens (terminals or nonterminal) exceeds the maximum capacity of the terminal or nonterminals area, a scroll-bar will appear and allow you to expose the hidden buttons and use them:

**BUILDING PRODUCTIONS** Productions with a common left-hand side symbol, are presented in AGG by a single object. We will refer to the object as
production and to each production within the object as a production line. A production in AGG is presented by a push button in the nonterminal area. When first created, a production is a single square in the drawing area. To build the right-hand side of production, click on any push-button representing a token and AGG will add an instance of that to token to last production-line of same production. To add new production-line, select Orelse from New menu or click on

Figure 35 production right hand side

the appropriate accelerator see figure(36). Once Orelse is selected, AGG draw a link between the left-hand side of the production and the proposed position for the first right-hand symbol of the new production-line. If you change your mind and decide not to add a new production-line, simply select Orelse again and the link will disappear. AGG directs all actions to the currently selected production.
To select a production, drag its push-button and drop it anywhere in the drawing area. If the production is already in the drawing area, click the mouse left button while the pointer positioned on the production left-hand symbol. The name of the currently selected production is always printed in the message area. To un-select a production, select another one or hide the production by selecting *Hide Selected Prd* from *Graph* menu or clicking on the appropriate accelerator, see figure(37).

To move a production to a new position, you can drag its push-button and drop it on the new position (the old object will automatically disappear) or press the mouse left button while the pointers positioned on the production left-hand side symbol and drag it to the new position.
THE SEMANTIC EDITOR  The semantic editor is a tool that allows you to associate functions with attributes, and link their arguments to some other attributes. To start the semantic editor, first select a production (you cannot start the editor before selecting a production) then select Semantic Editor from Edit menu or click on the appropriate accelerator. see figure(38).

Figure 38 attribute menu

Once started, the semantic editor loads the currently selected production and view it on its drawing area. Figure(39) shows the main components in a typical editing section
1. **Function area**: see section 2.1

2. **Control area**: Consists of the following set of *push-buttons*:
3. **Attributes area:** list all attributes associated with selected instance.

4. **Drawing area:** A canvas where the current production is displayed.

**ADDING SEMANTIC RULES** Using the semantic editor, you can easily select a function and associate it with a selected attribute. To select an attribute do the following steps.

1. Select a production line by clicking on any instance. Once a production-line is selected, the other lines will disappear from the screen figure(41.2).
2. Select the instance with which the attribute is associated. A set of push-bottons, each of which represents an attribute associated with selected instance, will appear on the attribute area.

3. Click on the push button representing the desired attribute. If the attribute is already associated with a rule, you will see the name and the arguments of the rule printed on the function area.

There are three type of rules that you can associate with an attribute:

1. a_rule: require a pre-declared function.

2. c_rule: uses the built-in copy function.

3. i_rule: prompt you for a string constant.

Click on the push-button labeled a_rule to associate a predefined function with the selected attribute, see figure(42).
A dialog with a list of all currently defined functions will appear and prompts you for a selection. Select one function and click on Apply button to associate it with the selected attribute.
Click on Show Type button to see the type of the argument and the return value of the function. If type-violation check is enabled, the type of the selected function must matches the type of the target attribute. Once the function is associated with an attribute, its name appears on the function area followed by an open bracket. At this stage, you can select attributes from different instances to associate with the function's arguments. To select attribute, simply click on any instance in the drawing area. All attributes associated with the instance will appear in squares.
underneath the instance, position the mouse pointer over one of the squares and release the left button to select the attribute. The name of the instance and its ordered (Sn) in the production, together with the name of the attribute, and its direction (Su or Sd) will automatically appear on the function area. The type of the selected attribute must matches the type of the argument. At this stage you can do one of the followings:

1. **Work on another attribute associated with the same instance:** simply click on the push-button labeled with that attribute name.

2. **Work on another attribute associated with a different instance in the same production line:** click on the push-button labeled *New Elm* and select the element and from the attribute list select the new attribute.

3. **Work on another attribute associated with a different instance in a different production-line:** click on the push-button labeled *New Line*, AGG will re-draw the whole production. Start the selection procedure again from the beginning.

**COPY AND INITIALIZE RULES** Copy-rule *c_rule* is a restricted form of *a_rule*, it uses the built-in one argument *copy* function. To select a copy rule click on the push-button labeled *c_rule* and select the appropriate attribute for the right-hand side of the function. To *initialize* an attribute to a constant value, click on the push-button labeled *i_rule*, a dialog will prompts you for the initial
value, type the input and click on OK. To work on another production, starts the semantic editor after selecting the new production.

FUNCTIONS AND LIBRARIES Before you can associate any function with attribute. You must supply the following information to AGG:

1. The type of the function.
2. The name of the function.
3. The type of each argument.

If the function you intend to use is defined in a library, simply load the library by selecting Load Lib from the Library menu. Otherwise, define your own function by selecting Add Function from Option menu. You will see the following dialog:

Figure 43 New-Function dialog

![Agg: New Function_popup]

Type the information in the following order:

1. The type of the function followed by a space.
2. The name of the function followed by an open bracket.

3. The type of each argument followed by a comma. The type of the last argument must be followed by a close bracket.

Click on OK to add the function. Figure(43) above, defines the function add to be of type num and have to arguments both are declared to be of type num.

MODIFYING TOKENS  AGG provides a set of editing tools that allows you to modify previously created tokens and their components. The changes on one token are propagated to the relevant parts of the structure using the shortest path available (since instances of token are grouped into a liked list rooted by the token itself, see chapter 3.2).

MODIFY NAME AND PATTERN  You can modify the name and the pattern associated with token by dragging the push button representing the object and drooping it on Modify Token push-button (the first button from the right in the control area).

![Figure 44 modify and delete tokens](image)

Modify Token.

Delete Token.

AGG will prompt you for the new name and pattern if the token is a terminal or just the new name if the token is a nonterminal. In both cases you can add
and define new attributes. If you select OK, the changes will propagates in
the internal structure and will be reflected on the graphical representation of the
attribute grammar.

**DELETING TOKENS** Deleting a token means deleting the template of the
token an all its instances. This action reflect both structural and graphical changes
on every production with one or more instances of the token. To delete a token
drag its push-button and drop it on *Delete Token* push-button (the first button from
the left in the control area) see figure(44). Before deleting the token, AGG marks
all the instants and perform a dynamic check. At this point you must inspect both
the graphical representation (all instances to be deleted are marked with crossed
lines), and the messages generated by the overall check, if you decided to go on
with the deletion select OK.

**DELETING AND INSERTING INSTANCES** You can schedule an instance
for deletion by clicking anywhere on its square in the drawing area, AGG will
marks the instance and performs a dynamic check. If the deletion cause a violation
to one of the integrity rules, such as left-recursion when the left-recursion check
is enabled, the task will be terminated and a warning message will be issued
explaining the reason behind the termination. You can select as many instances
as you want, if you changed your mind and decided not to delete one or more instances, simply select them again. To delete the selected instances, select *Delete Marked Elements* from *Edit* menu.

To insert an instance of a token, drag the token and drop it on the instance to left
or to the right of where you want to insert. You will see the following dialog:

Figure 47 insertion of instance

Select LEFT or RIGHT to insert the instance.
BIBLIOGRAPHY


VITA AUCTORIS

Daniel Ravan was born in 1961 in Baghdad-Iraq. He graduated from Salahdeen High School in 1980. From there he went on to the Mostansiria University where he obtained a B. Sc. in Mathematics and Statistics in 1985. He is currently a candidate for the Master's degree in Science at the University of Windsor and hopes to graduate in the Spring of 1995.