A VISUAL LANGUAGE FOR ADA PROGRAM UNIT SPECIFICATIONS

by

Christopher T. Gordon

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTERS OF SCIENCE

in

Computer Science

APPROVED:

S. Antoy, Chairman

C. Egyhazy

E. Haddad

May, 1990

Blacksburg, Virginia
A VISUAL LANGUAGE FOR ADA PROGRAM UNIT SPECIFICATIONS

by

Christopher T. Gordon

(ABSTRACT)

This thesis describes a visual programming language designed to describe and generate Ada program unit specifications.

The author first describes the foundations for the work, and gives a brief introduction to some of the features of the language.

Most of the thesis is dedicated to describing the visual representation for each portion of an Ada package specification. The BNF grammar of an Ada package specification is used as a basis for organization. By organizing the thesis via the package specification, all program unit specifications (i.e. package, task, subprogram and generic specifications) are described and given a representation in the language.

Toward the end of the thesis, the design and reference of a package specification is demonstrated in a hypothetical implementation.
ACKNOWLEDGEMENTS

I would first like to thank my chairman, Dr. Sergio Antoy. His was the first class that I attended at Virginia Tech, and he has been a constant source of guidance and friendship ever since then. I would also like to thank Dr. Egyhazy for his comradery and constant support during my graduate studies. Gratitude is also expressed to Dr. Haddad, who often gave me the opportunity to be involved in his research. Thanks also go to Ms. Norma Thomas for all of her help during my program at Virginia Tech.

But I feel that the most important debt of gratitude is to God and my family, who provided me with the emotional, spiritual and financial support that made my graduate schooling possible. Finally, I would like to thank all of my friends for their patience, support and prayers. Thank you, all of you, very much.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xx</td>
</tr>
<tr>
<td>1 INTRODUCTION AND BACKGROUND</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1.1 Motivation and Goals</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1.2 Evolution of the Notation for Representing Subprograms</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 The Lower Levels of Subprograms</td>
<td>10</td>
</tr>
<tr>
<td>1.2 The Representation for the Interdependencies of Data Types</td>
<td>12</td>
</tr>
<tr>
<td>1.2.1 Developing a Notation</td>
<td>12</td>
</tr>
<tr>
<td>1.2.2 The Final Notation</td>
<td>14</td>
</tr>
<tr>
<td>2 ORGANIZATION OF THESIS VIA THE PACKAGE SPECIFICATION</td>
<td>34</td>
</tr>
<tr>
<td>2.1 Assumptions</td>
<td>34</td>
</tr>
<tr>
<td>2.2 Top-down Organization of Thesis</td>
<td>35</td>
</tr>
<tr>
<td>3 OBJECT DECLARATION</td>
<td>39</td>
</tr>
<tr>
<td>3.1 Simple Object Declarations</td>
<td>39</td>
</tr>
<tr>
<td>3.1.1 Constraints</td>
<td>43</td>
</tr>
<tr>
<td>3.2 Objects of Constrained Array Type</td>
<td>45</td>
</tr>
<tr>
<td>4 NUMBER DECLARATION</td>
<td>56</td>
</tr>
<tr>
<td>5 TYPE DECLARATION</td>
<td>59</td>
</tr>
<tr>
<td>5.1.1 BNF and Organization of Type Declarations</td>
<td>59</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>5.1.2 Discriminant Part</td>
<td>60</td>
</tr>
<tr>
<td>5.2 Type Definition</td>
<td>60</td>
</tr>
<tr>
<td>5.2.1 Enumeration Type Definition</td>
<td>61</td>
</tr>
<tr>
<td>5.2.2 Integer Type Definition</td>
<td>62</td>
</tr>
<tr>
<td>5.2.3 Real Type Definition</td>
<td>62</td>
</tr>
<tr>
<td>5.2.4 Array Type Definition</td>
<td>63</td>
</tr>
<tr>
<td>5.2.5 Record Type Definition</td>
<td>64</td>
</tr>
<tr>
<td>5.2.6 Access Type Definition</td>
<td>67</td>
</tr>
<tr>
<td>5.2.7 Derived Type Definition</td>
<td>67</td>
</tr>
<tr>
<td>5.3.1 Incomplete Type Declaration</td>
<td>68</td>
</tr>
<tr>
<td>5.3.2 Private Type Declaration</td>
<td>69</td>
</tr>
<tr>
<td>6 SUBTYPE DECLARATION</td>
<td>92</td>
</tr>
<tr>
<td>7 SUBPROGRAM DECLARATION</td>
<td>96</td>
</tr>
<tr>
<td>7.1 BNF and Organization</td>
<td>96</td>
</tr>
<tr>
<td>7.2.1 High Level of Subprogram Specifications</td>
<td>97</td>
</tr>
<tr>
<td>7.2.2 Low Level of Subprogram Specifications</td>
<td>98</td>
</tr>
<tr>
<td>7.3 Operations Performable on the Subprogram Table</td>
<td>98</td>
</tr>
<tr>
<td>8 PACKAGE DECLARATION</td>
<td>108</td>
</tr>
<tr>
<td>9 TASK DECLARATION</td>
<td>111</td>
</tr>
<tr>
<td>9.1 BNF and Basic Notation</td>
<td>111</td>
</tr>
<tr>
<td>9.2 Task Types</td>
<td>111</td>
</tr>
<tr>
<td>9.3 Entry Declarations</td>
<td>112</td>
</tr>
<tr>
<td>10 GENERIC DECLARATION</td>
<td>118</td>
</tr>
<tr>
<td>11 EXCEPTION DECLARATION</td>
<td>123</td>
</tr>
<tr>
<td>12 GENERIC INSTANTIATION</td>
<td>125</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 RENAMING DECLARATION</td>
<td>129</td>
</tr>
<tr>
<td>14 DEFERRED CONSTANT DECLARATION</td>
<td>131</td>
</tr>
<tr>
<td>15 REPRESENTATION CLAUSE</td>
<td>133</td>
</tr>
<tr>
<td>16 USE CLAUSE</td>
<td>135</td>
</tr>
<tr>
<td>17 AN EXAMPLE OF THE DESIGN OF A PACKAGE SPECIFICATION IN A</td>
<td>137</td>
</tr>
<tr>
<td>HYPOTHETICAL IMPLEMENTATION</td>
<td></td>
</tr>
<tr>
<td>18 AN EXAMPLE OF THE REFERENCE OF A PACKAGE SPECIFICATION IN</td>
<td>163</td>
</tr>
<tr>
<td>A HYPOTHETICAL IMPLEMENTATION</td>
<td></td>
</tr>
<tr>
<td>19 CONCLUSIONS</td>
<td>170</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>171</td>
</tr>
<tr>
<td>VITA</td>
<td>159</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1.1</td>
<td>Goguen's notation applied to the ADT of nat</td>
</tr>
<tr>
<td>1.2</td>
<td>Goguen's notation applied to the ADT of set</td>
</tr>
<tr>
<td>1.3</td>
<td>An attempt to use Goguen's notation to describe the package of Table 1.1</td>
</tr>
<tr>
<td>1.4</td>
<td>First notation describing the package of Table 1.1</td>
</tr>
<tr>
<td>1.5</td>
<td>Key to first, second and third notations</td>
</tr>
<tr>
<td>1.6</td>
<td>Second notation describing the package of Table 1.1</td>
</tr>
<tr>
<td>1.7</td>
<td>Third notation describing the package of Table 1.1</td>
</tr>
<tr>
<td>1.8</td>
<td>Tabular notation describing the package of Table 1.1</td>
</tr>
<tr>
<td>1.9</td>
<td>Lower level of the second occurrence of function LOCATE in Table 1.1</td>
</tr>
<tr>
<td>1.10</td>
<td>Lower levels of the second and third occurrences of function &quot;&amp;&quot; in Table 1.1</td>
</tr>
<tr>
<td>1.11</td>
<td>An attempt to use a tabular notation to describe the type dependencies of Table 1.2</td>
</tr>
<tr>
<td>1.12</td>
<td>Iconic notation describing the type dependencies of Table 1.2</td>
</tr>
<tr>
<td>1.13</td>
<td>Iconic notation describing the package of Table 1.3</td>
</tr>
<tr>
<td>1.14</td>
<td>Iconic notation describing the package of Table 1.4</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.1.a</td>
<td>Notation for the high level of a variable declaration</td>
</tr>
<tr>
<td>3.1.b</td>
<td>Notation for the high level of a constant declaration</td>
</tr>
<tr>
<td>3.1.c</td>
<td>Notation for the low level of a variable declaration</td>
</tr>
<tr>
<td>3.1.d</td>
<td>Notation for the low level of a constant declaration</td>
</tr>
<tr>
<td>3.2.a</td>
<td>Notation for a range constraint defined using a range identifier</td>
</tr>
<tr>
<td>3.2.b</td>
<td>Notation for a range constraint defined using two simple expressions</td>
</tr>
<tr>
<td>3.3.a</td>
<td>Example of range constraint which is defined using enumeration type SUMMER</td>
</tr>
<tr>
<td>3.3.b</td>
<td>Example of range constraint which is defined using simple expression JUNE and simple expression AUGUST</td>
</tr>
<tr>
<td>3.4.a</td>
<td>Notation for a floating point constraint</td>
</tr>
<tr>
<td>3.4.b</td>
<td>Notation for a fixed point constraint</td>
</tr>
<tr>
<td>3.5.a</td>
<td>Example of a floating point constraint</td>
</tr>
<tr>
<td>3.5.b</td>
<td>Example of a fixed point constraint</td>
</tr>
<tr>
<td>3.6</td>
<td>Notation for an index constraint</td>
</tr>
<tr>
<td>3.7</td>
<td>Example of an index constraint</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.8.a</td>
<td>Notation for the high level of a discriminant constraint applied to a subtype declaration</td>
</tr>
<tr>
<td>3.8.b</td>
<td>Notation for the low level of a discriminant constraint</td>
</tr>
<tr>
<td>3.9.a</td>
<td>Notation for the high level of a variable of constrained array type declaration</td>
</tr>
<tr>
<td>3.9.b</td>
<td>Notation for the high level of a constant of constrained array type declaration</td>
</tr>
<tr>
<td>3.9.c</td>
<td>Notation for the low level of a variable of constrained array type declaration</td>
</tr>
<tr>
<td>3.9.d</td>
<td>Notation for the low level of a constant of constrained array type declaration</td>
</tr>
<tr>
<td>3.10.a</td>
<td>Example of the high level of constrained array object COLOR_TABLE</td>
</tr>
<tr>
<td>3.10.b</td>
<td>Example of the low level of constrained array object COLOR_TABLE from:</td>
</tr>
<tr>
<td></td>
<td>COLOR_TABLE : array(1 .. N) of COLOR;</td>
</tr>
<tr>
<td>4.1.a</td>
<td>Icon for a number constant declaration</td>
</tr>
<tr>
<td>4.1.b</td>
<td>Notation for the low level of a number constant declaration</td>
</tr>
<tr>
<td>4.2.a</td>
<td>Example of the icon of the number constant PI</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>4.2.b Example of the low level of the number constant PI</td>
<td>58</td>
</tr>
<tr>
<td>5.1.a Notation for the high level of a declared type with a discriminant</td>
<td>71</td>
</tr>
<tr>
<td>5.1.b Notation for the low level of a declared type with a discriminant</td>
<td>71</td>
</tr>
<tr>
<td>5.2.a Icon for an enumeration type declaration</td>
<td>72</td>
</tr>
<tr>
<td>5.2.b Notation for the low level of an enumeration type declaration</td>
<td>72</td>
</tr>
<tr>
<td>5.3.a Example of the icon of enumeration type COLORS</td>
<td>73</td>
</tr>
<tr>
<td>5.3.b Example of the low level of the enumeration type COLORS from:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type COLORS is (RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET);</td>
</tr>
<tr>
<td>5.4.a Icon for an integer type declaration</td>
<td>74</td>
</tr>
<tr>
<td>5.4.b Notation for the low level of an integer type declaration</td>
<td>74</td>
</tr>
<tr>
<td>5.5.a Example of the icon of the integer type PAGE_NUM</td>
<td>75</td>
</tr>
<tr>
<td>5.5.b Example of the low level of the integer type PAGE_NUM from:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type PAGE_NUM is range 1 .. 200;</td>
</tr>
<tr>
<td>5.6.a Icon for a real type declaration</td>
<td>76</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>5.6.b</td>
<td>Notation for the low level of a floating point type declaration</td>
</tr>
<tr>
<td>5.6.c</td>
<td>Notation for the low level of a fixed point type declaration</td>
</tr>
<tr>
<td>5.7.a</td>
<td>Example of the low level of floating point type MASS from:</td>
</tr>
<tr>
<td></td>
<td>type MASS is digits 5 range 0.0 .. 1.0;</td>
</tr>
<tr>
<td>5.7.b</td>
<td>Example of the low level of fixed point type VOLTAGE from:</td>
</tr>
<tr>
<td></td>
<td>type VOLTAGE is delta 0.1 range -12.0 .. 24.0;</td>
</tr>
<tr>
<td>5.8.a</td>
<td>Icon for an array type declaration</td>
</tr>
<tr>
<td>5.8.b</td>
<td>Notation for the high level of an unconstrained array type declaration</td>
</tr>
<tr>
<td>5.8.c</td>
<td>Notation for the low level of an unconstrained array type declaration</td>
</tr>
<tr>
<td>5.9.a</td>
<td>Example of the icon of unconstrained array type VECTOR</td>
</tr>
<tr>
<td>5.9.b</td>
<td>Example of the high level of unconstrained array type VECTOR</td>
</tr>
<tr>
<td>5.9.c</td>
<td>Example of the low level of unconstrained array type VECTOR from: type VECTOR is array(INTEGER range &lt;&gt;) of REAL;</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>5.10.a</td>
<td>Icon for a record type declaration</td>
</tr>
<tr>
<td>5.10.b</td>
<td>Notation for a record type declaration with a component list of null</td>
</tr>
<tr>
<td>5.10.c</td>
<td>Notation for the high level of a record type declaration without a variant part</td>
</tr>
<tr>
<td>5.10.d</td>
<td>Notation for the low level of a record type declaration without a variant part</td>
</tr>
<tr>
<td>5.11.a</td>
<td>Example of the high level of the record type DATE</td>
</tr>
<tr>
<td>5.11.b</td>
<td>Example of the low level of the record type DATE from:</td>
</tr>
</tbody>
</table>

```plaintext
type DATE is
  record
    DAY : INTEGER range 1 .. 31;
    MONTH : MONTH_NAME;
    YEAR : INTEGER range 0 .. 4000;
  end record;
```

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.12.a</td>
<td>Notation for the high level of a record type declaration with a variant part</td>
<td>82</td>
</tr>
<tr>
<td>5.12.b</td>
<td>Notation for the low level of a record type declaration with a variant part</td>
<td>82</td>
</tr>
<tr>
<td>5.13.a</td>
<td>Example of the high level of the record type PERIPHERAL (which has a variant part)</td>
<td>83</td>
</tr>
<tr>
<td>5.13.b</td>
<td>Example of the low level of the record type PERIPHERAL (which has a variant part) from:</td>
<td>83</td>
</tr>
</tbody>
</table>
LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>type P$$\text{E}$$RI$$\text{P}$$HERAL(UNIT : DEVICE := DISK) is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>record</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STATUS : STATE;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>case UNIT is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>when PRINTER =&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LINE_COUNT : INTEGER range 1 .. PAGE_SIZE;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>when others =&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CYLINDER : CYLINDER_INDEX;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRACK : TRACK_NUMBER;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end case;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end record;</td>
<td>84</td>
</tr>
<tr>
<td>5.14.a</td>
<td>Icon for an access type declaration</td>
<td>85</td>
</tr>
<tr>
<td>5.14.b</td>
<td>Notation for the low level of an access type declaration</td>
<td>85</td>
</tr>
<tr>
<td>5.15.a</td>
<td>Example of the icon of the access type STUDENT POINTER</td>
<td>86</td>
</tr>
<tr>
<td>5.15.b</td>
<td>Example of the low level of the access type STUDENT POINTER</td>
<td>86</td>
</tr>
<tr>
<td>5.16</td>
<td>The set of icons denoting derived type declarations</td>
<td>87</td>
</tr>
<tr>
<td>5.17.a</td>
<td>Icon for a derived type declaration</td>
<td>88</td>
</tr>
<tr>
<td>5.17.b</td>
<td>Notation for the low level of a derived type declaration</td>
<td>88</td>
</tr>
<tr>
<td>5.18.a</td>
<td>Example of the icon of the derived type FAHRENHEIT</td>
<td>89</td>
</tr>
<tr>
<td>5.18.b</td>
<td>Example of the low level of the derived type FAHRENHEIT</td>
<td>89</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.19</td>
<td>The set of icons denoting private type declarations</td>
<td>90</td>
</tr>
<tr>
<td>5.20</td>
<td>The set of icons denoting limited private type declarations</td>
<td>91</td>
</tr>
<tr>
<td>6.1</td>
<td>Various symbols for subtypes</td>
<td>93</td>
</tr>
<tr>
<td>6.2.a</td>
<td>Symbol for the high level of a subtype declaration</td>
<td>94</td>
</tr>
<tr>
<td>6.2.b</td>
<td>Notation for the low level of a subtype declaration</td>
<td>94</td>
</tr>
<tr>
<td>6.3.a</td>
<td>Example of the icon of the subtype SIZE</td>
<td>95</td>
</tr>
<tr>
<td>6.3.b</td>
<td>Example of the low level of the subtype SIZE</td>
<td>95</td>
</tr>
<tr>
<td>7.1.a</td>
<td>Notation for the high level of a procedure with no parameters</td>
<td>100</td>
</tr>
<tr>
<td>7.1.b</td>
<td>Notation for the high level of a procedure with assorted parameters</td>
<td>100</td>
</tr>
<tr>
<td>7.1.c</td>
<td>Notation for the high level of a function with no parameters</td>
<td>100</td>
</tr>
<tr>
<td>7.1.d</td>
<td>Notation for the high level of a function with assorted parameters</td>
<td>100</td>
</tr>
<tr>
<td>7.2.a</td>
<td>Example of the high level of a procedure with no parameters</td>
<td>101</td>
</tr>
<tr>
<td>7.2.b</td>
<td>Example of the high level of a procedure with assorted parameters</td>
<td>101</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.2.c</td>
<td>Example of the high level of a function with no parameters</td>
<td>101</td>
</tr>
<tr>
<td>7.2.d</td>
<td>Example of the high level of a function with parameters</td>
<td>101</td>
</tr>
<tr>
<td>7.3.a</td>
<td>Notation for the low level of a procedure</td>
<td>102</td>
</tr>
<tr>
<td>7.3.b</td>
<td>Notation for the low level of a function</td>
<td>102</td>
</tr>
<tr>
<td>7.4.a</td>
<td>Example of the low level of a procedure with no parameters</td>
<td>103</td>
</tr>
<tr>
<td>7.4.b</td>
<td>Example of the low level of a procedure with parameters</td>
<td>103</td>
</tr>
<tr>
<td>7.4.c</td>
<td>Example of the low level of a function with no parameters</td>
<td>103</td>
</tr>
<tr>
<td>7.4.d</td>
<td>Example of the low level of a function with parameters</td>
<td>103</td>
</tr>
<tr>
<td>7.5.a</td>
<td>The two subprograms SORT_LIST and EQUAL are marked to be joined</td>
<td>104</td>
</tr>
<tr>
<td>7.5.b</td>
<td>The result of joining the subprograms of Figure 7.5.a</td>
<td>104</td>
</tr>
<tr>
<td>7.6</td>
<td>The result of yanking rows from the table in Figure 1.8</td>
<td>105</td>
</tr>
<tr>
<td>7.7</td>
<td>The result of yanking columns from the table in Figure 7.6</td>
<td>106</td>
</tr>
<tr>
<td>7.8</td>
<td>The result of condensing overloaded subprograms in the table of Figure 7.7</td>
<td>107</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.1</td>
<td>Notation for the highest level of a package specification</td>
<td>110</td>
</tr>
<tr>
<td>9.1</td>
<td>Icon for a task declaration</td>
<td>114</td>
</tr>
<tr>
<td>9.2</td>
<td>Icon for a task type declaration</td>
<td>115</td>
</tr>
<tr>
<td>9.3.a</td>
<td>Notation for the high level of an entry declaration</td>
<td>116</td>
</tr>
<tr>
<td>9.3.b</td>
<td>Notation for the low level of an entry declaration</td>
<td>116</td>
</tr>
<tr>
<td>9.4.a</td>
<td>Example of the high level of entry declaration READ</td>
<td>117</td>
</tr>
<tr>
<td>9.4.b</td>
<td>Example of the low level of entry declaration READ from: entry READ (N : in INDEX; V : out ITEM);</td>
<td>117</td>
</tr>
<tr>
<td>10.1.a</td>
<td>Icon for a generic package declaration</td>
<td>121</td>
</tr>
<tr>
<td>10.1.b</td>
<td>Notation for the high level of a generic subprogram declaration</td>
<td>121</td>
</tr>
<tr>
<td>10.2.a</td>
<td>Notation for the high level of a generic package and its generic parameters</td>
<td>122</td>
</tr>
<tr>
<td>10.2.b</td>
<td>Notation for the low level of a generic package and its generic parameters</td>
<td>122</td>
</tr>
<tr>
<td>11.1.a</td>
<td>Icon for an exception declaration</td>
<td>124</td>
</tr>
<tr>
<td>11.1.b</td>
<td>Example of the icon of exception OVERFLOW</td>
<td>124</td>
</tr>
<tr>
<td>12.1.a</td>
<td>Notation for the high level of the instantiation of a generic</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>12.1.b Notation for the low level of the instantiation of a generic package</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>12.2.a Notation for the high level of the instantiation of a generic subprogram</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>12.2.b Notation for the low level of the instantiation of a generic subprogram</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>13.1.a Notation for the renaming of a variable</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>13.1.c Notation for the renaming of a package</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>13.1.d Notation for the renaming of a subprogram</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>14.1.a Icon for a private constant</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>14.1.b Icon for a limited private constant</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>16.1 Notation for depicting the use clause</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>17.1 A package specification library window</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>17.2 Dragging the package icon into the package specification library window</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>17.3 The text window requesting the name of the newly created package specification</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>17.4 Selection of the number constant object</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>17.5</td>
<td>The number constant MAX</td>
<td>148</td>
</tr>
<tr>
<td>17.6</td>
<td>Selection of the private array type</td>
<td>149</td>
</tr>
<tr>
<td>17.7</td>
<td>Selection of the predefined type FLOAT</td>
<td>150</td>
</tr>
<tr>
<td>17.8</td>
<td>Access of the arc operation</td>
<td>151</td>
</tr>
<tr>
<td>17.9</td>
<td>Use of the arc to denote the dependency of type STACK on type FLOAT</td>
<td>152</td>
</tr>
<tr>
<td>17.10</td>
<td>The selection of the discrete subtype indication constraint</td>
<td>153</td>
</tr>
<tr>
<td>17.11</td>
<td>The use of copy operation to input the upper bound of the index constraint</td>
<td>154</td>
</tr>
<tr>
<td>17.12</td>
<td>Selection of the subprogram</td>
<td>155</td>
</tr>
<tr>
<td>17.13</td>
<td>Use of the copy operation to define the first formal parameter to be of type STACK</td>
<td>156</td>
</tr>
<tr>
<td>17.14</td>
<td>Use of the copy operation to define the second formal parameter to be of type FLOAT</td>
<td>157</td>
</tr>
<tr>
<td>17.15</td>
<td>Determining modes of the formal parameters</td>
<td>158</td>
</tr>
<tr>
<td>17.16</td>
<td>Accessing the low level of procedure PUSH</td>
<td>159</td>
</tr>
<tr>
<td>17.17</td>
<td>Name of the first formal parameter and request to input the name of the second formal parameter</td>
<td>160</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>17.18  Use of Exceptions mouse button</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>17.19  Final visual representation of STACK_PACK package</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>18.1   The package specification library window containing the package STACK_PACK</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>18.2   Accessing the low level of object MAX</td>
<td>166</td>
<td></td>
</tr>
<tr>
<td>18.3   Accessing the low level of type STACK</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>18.4   Accessing the low level of procedure PUSH</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>18.5   Fully expanded view of package STACK_PACK</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Specification for package TEXT_HANDLER</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>Specification for package ALBUMS</td>
<td>27</td>
</tr>
<tr>
<td>1.2</td>
<td>Specification for package LINKED_LIST</td>
<td>30</td>
</tr>
<tr>
<td>1.3</td>
<td>Specification for package MUTUALLY_DEPENDENT</td>
<td>32</td>
</tr>
<tr>
<td>17.1</td>
<td>Specification for package STACK_PACK</td>
<td>143</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

1.1.1.1 Motivation and Goals

Recently, CASE (Computer Aided Software Engineering) has become very popular. CASE technology attempts to ease the software engineer's task by exploiting powerful workstation hardware, graphics and software development methodologies.

The use of visual programming (i.e. the use of graphics to visually represent software) is an integral part of any CASE tool. Visual representations of programs are considered to be easier and quicker for programmers to understand, since the information is represented in a two-dimensional form, a form much more understandable to the human mind. Textual representations of programs (written code) are, by contrast, considered to be only one dimensional, since they consist of a single string of characters (the indentation of blocks of code is an attempt to impose some amount of a "second dimension" on this single dimension representation) [Chang89b].

Also, the development of programs may be eased, since it may no longer be necessary for the programmer to type all of the code. Keywords, reserved words and other elements of the language may be inferred from the diagrams and automatically inserted into the code which is generated from the diagrams.

It is important to mention that there are two ways that program visualization can be accomplished: static visualization and
dynamic visualization. Static visualization refers to the use of two-dimensional notations to represent programs. Dynamic visualization refers to the use of two-dimensional notations to display the actions of algorithms and data during execution. The visual language described in this thesis is static [Myers89].

The goal in writing this thesis is to develop a CASE tool for the Ada programming language. This tool will be a visual language and its purpose will be to visually describe all of the specifications for Ada program units.

Specifications are the portions of Ada program units that specify which entities of the unit may be used by other programmers. Ada program unit specifications are primarily intended for human consumption; therefore, attempts should be made to make them easier for humans to understand.

Ada supports the notion of modularity and data abstraction. Modularity refers to the methods of describing programs, not as single entities, but as being comprised of many separate program-like units (e.g. packages, procedures and functions). For the sake of brevity, we will describe data abstraction as a method of ensuring that programmers use the appropriate interfaces when using the program units of other programmers.

Because these two notions are enforced in Ada and because program unit specifications are intended to function as a type of documentation for humans, programmers must often refer to the specifications of other programmers to find useful software which
they can use in their own code. Therefore, the Ada specification is constantly being referenced by a person other than the original author. It is for this reason that a visual representation of program unit specifications will prove useful for the Ada programming community.

1.1.1.2 Evolution of the Notation for Representing Subprograms

In retrospect, the inspiration for this visual language came when the observation was made that a graphic notation from the field of abstract data types could perhaps be used to describe Ada subprogram declarations. The graphic notation had been developed by J. A. Goguen, J. W. Thatcher, and E. G. Wagner [Goguen78]. Goguen and his colleagues had, among other things, proposed a graphical system to represent signatures for operations performed on abstract data types. These ideas are closely linked to algebraic specifications.

Thomas Edison once stated that invention is one percent inspiration and ninety-nine percent perspiration. As stated before, the visual description of subprogram declarations was inspired by the work of Goguen. The other parts (mainly type declarations) represent the "perspiration". Both portions required a great deal of thought and work to develop. Yet, the former was based on a somewhat theoretical foundation which guided the thinking processes. The latter was not, and therefore demanded the independent exploration of many ideas.
Although the method of describing subprogram declarations was the "inspired" portion of this thesis, it was not developed without a cost in time and energy. This section of the thesis is devoted to explaining the evolutionary processes which resulted in the present form of the visual language. This is done so that the reader might understand how the ideas and notations in this thesis originated, the reasons for their present forms, and the thought that went into developing them (although we will refrain from describing every transitional form in this evolutionary process). Also, this background will give the reader a preview of the visual language which will be described in detail in subsequent chapters.

Let us begin by describing some ideas pertaining to abstract data types and algebraic specifications. The algebraic specification for the natural numbers is shown in [Antoy89] and is defined as follows:

\begin{verbatim}
sort nat
  constructors
    0;
    succ(nat).
\end{verbatim}

This means that a natural number is either zero or the successor of a natural number. Goguen's representation of the signature of the algebraic specification for the natural numbers is shown in Figure 1.1 [Goguen78].

Notice that, in Figure 1.1, "nat" (abbreviation for natural number) is placed in a "bubble". These "bubbles" represent data
types. "Successor" is a function which takes, as input, a natural number and returns a new natural number (the successor of the input number). Functions are represented by dots. Notice, too, that the arcs denote the movement of data through the functions. The arrowhead is placed to denote the returned value and the "direction of travel" along the arcs.

In Figure 1.1, the use of constants is introduced in the form of the constant zero. A constant can be thought of as a function with no arguments. If the reader inspects the illustration, he will find that the notation is consistent with this interpretation of a constant.

Figure 1.2 shows a representation for the data type "set" [Goguen78]. There are a couple of important concepts in this example.

A concept which should be mentioned here is the use of multiple parameters. For example, "equal" takes two sets and returns a boolean value. We see that the parameters "flow" together and that a function always returns a single value. Goguen's notation does not allow side effects - all functions return one and only one value.

Also notice the abstraction in Figure 1.2. Only what is necessary is stated. Issues about the ordering of parameters are irrelevant at this level of detail. These graphs do not imply any such order, since positional association is irrelevant in this notation.

When work began on this thesis, the first goal was to use
Goguen's notation to depict Ada subprogram specifications using Goguen's graphs. Skeletal code would then be generated from the graphs and compiled. The graphs would be as expressive as possible, in order to generate code that was as complete as possible.

Early experimentation proved that Goguen's graphs, while useful for small declarations, were simply not practical for any realistic specifications.

Table 1.1 shows an Ada package specification from the Ada Reference Manual [Gehani84]. This specification will be referred to throughout this section. It has been chosen as a somewhat complicated example which will help the reader realize the merit of these ideas for nontrivial cases.

Figure 1.3 is an attempt to use Goguen's notation for the code in Table 1.1, although this illustration shows only functions of the specification. Although Goguen's notation does not incorporate the notion of procedures (which do not return values), adaptations can be made to facilitate their use. Notice also that not all of the boolean functions have been included ("=", "<", "<=", etc.). Regardless of these omissions, this graph will be sufficient to demonstrate certain points. Although incomplete, the graph seems to be less intelligible than the Ada code which was shown earlier. Of course, this point may be debatable, but the curved arcs, combined with the tendency of arcs to cross, seem to make the graph less comprehensible. The completion of this graph would result in the addition of more bubbles and arcs, which would further complicate the graph.
The conclusion was that most of the complication came from the curvature of the arcs rather than their crossing. It was reasoned that crossing lines occur in road maps, graphs, and electrical diagrams, but crossing arcs seem to "flow" together, causing the eye to become "lost". As a result, an attempt was made to borrow ideas from electrical diagrams: The arcs were straightened and the bubbles were changed to rectangles. Functions would be represented by large, black triangles, the triangle denoting the direction of travel of data through the function. The resulting graph which describes the specification is depicted in Figure 1.4.

Again, it is not necessary to complete the graph to realize that there are still flaws with the practical use of this type of graph. First, the large number of crossings of lines does have a detrimental effect on the graph. Second, since type "text" is used so frequently, we will run out of room with the "text" rectangle. Of course, the rectangle could possibly be enlarged whenever it was necessary for more parameters or return values, but it was felt that this would have a detrimental effect on the aesthetics and, consequently, the readability of the notation.

After some thought, it was decided to represent all types as a vertical bar and maintain the other notations that had been evolved. The type bars would have the name of the type on top and bottom to improve readability. Subprograms would be represented along the horizontal axis. Large empty (white) arrows represented procedures; while large, black arrows represented functions as in
the previous notation. Parameters would be represented by lines from the appropriate type bar to the subprogram arrow. A small arrowhead was placed at the point where the horizontal parameter line intersected the vertical type bar. Small, empty arrowheads pointed toward the subprogram indicated an "in" parameter. Small, empty arrowheads pointed toward the type bar indicated an "out" parameter. Two small, empty arrowheads pointed in opposite directions indicated an "in out" parameter. Finally, a small black arrowhead pointed toward a type bar indicated a returned value from a function. A key to these symbols is shown in Figure 1.5. The result of using this type of representation is shown in Figure 1.6. The notation has been applied to the specification of Table 1.1.

Again, the profusion of lines seems to make the graph difficult to read. It was observed that all inputs can be placed on one line, rather than using many separate lines to denote the parameters of a single subprogram declaration. With a few exceptions, every subprogram could be described using a single straight line (the exceptions arose because of the spatial placement of type bars and the use of two parameters of the same type). These ideas are demonstrated in Figure 1.7.

The observation was made that the form of this notation is similar to a matrix. Thus, it was concluded that the specification of a group of subprograms is largely representable as a matrix-like structure. Of course, this matrix does not include all of the information of the specification, but does represent the highest
level of abstraction.

An alternate, "cleaner", representation for this matrix is one similar to a spreadsheet. This is actually a table and we will refer to it as such for the rest of the paper. Using a table, parameters can be denoted using the previously mentioned arrowhead convention. Parameters for a subprogram are represented by placing the appropriate type of arrowhead in the cell at the intersection of the subprogram row and type column.

As stated before, much of the information in the Ada code is not represented in the table format; this information is hidden. This is a form of abstraction and is considered a good tool in aiding comprehension. The names of the parameters and their order within the subprogram call are irrelevant at this high level. By freeing the maintainer of a program from such irrelevant information we allow him to concentrate only on what is necessary for a basic understanding of the specification: the interaction of subprograms and types. Lower levels of abstraction will be dealt with later in the thesis. Figure 1.8 shows the resulting representation of our specification.

It ought to be mentioned here that the thinking process has come full circle. All of these methods for representing the specifications of subprograms are equivalent. It is well known that sparse matrices are representable as graphs and that graphs are representable as adjacency matrices [Aho74]. Although any of the notations mentioned can be used to represent the specification, the
table method has the elegance and readability which Goguen's graphs have for small specifications. Also, there seems to be a slight difference in emphasis when contrasting the two notations. One might argue that Goguen's notation places emphasis on the data types, while this tabular notation achieves a balance between the description of subprograms and data types.

Notice that an effort is made to group overloaded subprograms together via shading. In this manner, all instances of an overloaded subprogram can be viewed, yet they are grouped so that all instances can be observed. On systems with adequate hardware, color may be used to make such distinctions.

Since we have described a notation for visually describing the specification, and since this notation is similar to a spreadsheet, it will be useful to access it like a spreadsheet. In a later section, a detailed description of desired spreadsheet-like operations will be given.

By using the pointing device on the table, one will be able to access lower levels of abstraction (e.g. parameter names, order of parameters). This is the subject of the next section.

1.1.2 The Lower Levels of Subprograms

When we refer to subprogram specifications in the visual language, lower levels of abstraction will be reachable only through the above table representation in an effort to enforce abstraction. The system will be designed for use with a pointing device such as a mouse, although provisions might be made for other types of input
devices. By using the pointing device at the table level, lower levels of abstracion will be made accessible.

For example, if a user desired more information about a subprogram, the pointer could be placed over the name of the subprogram and the mouse button would be clicked. A small table would appear to show the order and names of the parameters. The subprograms would now have been fully specified. From this lower level the user could exit the visual language and enter the editor to modify the actual subprogram specification code.

An example from the previously shown package might be the second occurrence of the LOCATE function. Figure 1.9 shows the result of viewing this function at a lower level.

Notice that the format of this smaller table in Figure 1.9 is similar to that of the higher level. However, the order of the types has changed. This new order reflects the order of the parameters in the function specification. Also notice that the names of passed parameters are displayed under the parameters.

One may have noticed that the operator "&" is overloaded. If the reader looks at the high level table closely, he may notice that the second and third occurrence of this function seem to be equivalent. However, by examining the lower levels of each, one would see the structures of Figure 1.10.

Since the structures of Figure 1.10 are in the appropriate order, we can contrast the two occurrences. We see that this case of overloading allows the function to be called without concern for the
placement of parameters, since the only difference in the structures is the placement of types.

One might argue that in cases such as these we should combine the two into one table, and adopt a special convention to indicate that the order of parameters is irrelevant. There are several arguments against this idea. The first is one of principle: The two functions are separate entities, and should be represented as such. The second is one of theoretical concern; the two examples above may perform entirely different computations (although a programmer who would commit such a horrible stylistic error should have his keyboard broken). Similar formats do not guarantee similar semantics. The third is one of pragmatics: To do so would needlessly complicate the language and its implementation.

We have informally demonstrated how to fully specify the Ada subprogram. In a future section these methods will be shown in a more formal manner, using the Backus-Naur form grammar.

1.2 The Representation for the Interdependencies of Data Types

The notations developed above are useful for specifying subprograms. However, there are two main components of most Ada package specifications: subprogram specifications and data type declarations. In order to fully represent a package specification it is necessary to deal with the latter component.

1.2.1 Developing a Notation
Since the table notation seemed to do such an efficient job of defining subprograms, it was hoped that it would be possible to modify the tabular notation so that type dependencies could be demonstrated using a similar notation. By type dependencies, we mean the usage of types to define other types. Unfortunately, all efforts to adapt the table notation to represent type dependencies were unsuccessful.

Table 1.2 shows a package specification taken from [Booch87]. The specification contains type declarations to be used in a record store database and is named "Albums". Notice that this type contains derived types, record types, an enumeration type, a subtype and an array type.

Figure 1.11 shows the use of the table notation to describe this specification. Notice that the use of symbols and icons has been adopted to denote data structures (e.g. arrays, records, pointers, etc.). To determine type dependencies, one follows the row of the type in question. The choice of icon in a column indicates the kind of dependency or relationship between the type of the row and the type of the column. By comparing the code of Table 1.2 with Figure 1.11 and its corresponding key, the symbols and icons should be apparent. Notice, too, that those declarations which do not make use of typemarks (e.g. a declaration of an integer) have columns labeled with a symbol, rather than a typemark.

There are several problems with this notation. Notice that the declarations form a curve along the diagonal of the table. This is
because, as a new row is added to define a new type, a new column must next be added if later type declarations are to utilize this new type. The set of types grows as new types are added, unlike the set of parameter types which we will assume does not grow during or after the subprograms are defined. (Although, in fact, more types can be declared after the declaration of subprograms, this assumption will not adversely affect the generality of the language, since there is no way that a type declaration can depend upon the declaration of a subprogram.) Consequently, any time type declarations make use of previous declarations, this diagonal phenomenon will occur. It seems to be wasteful of space as well as aesthetically unpleasing.

1.2.2 The Final Notation

It was believed that a better notation for the interdependencies of type would be one which was similar to Figure 1.7. Iconic elements could be incorporated into this type of diagram to denote the data structures being declared. Figure 1.12 will show the declarations of Table 1.2 using the new notation.

The lines and arrows in this notation specify the "built from" nature of types. For example, "SONG" in Figure 1.12 is built from the "LENGTH" and "NAME" type. If an arc is depicted as emanating from a type (the arc is drawn intersecting the type box and there is no inbound arrow), that type is being used to build another type.

The code in Table 1.2 does not give us any indication of how access types and types which contain access types are handled.
Thus, two other package specification examples have been included.

Table 1.3 shows a package specification for a linked list from [Gehani84]. The package contains an access type which points to a record type which contains two fields of the access type. Figure 1.13 shows the corresponding diagram. Notice that the access type is denoted by what appears to be a large arrow (or "pointer").

Table 1.4 shows code for mutually dependent data structures. This code is also taken from [Gehani84]. In this specification there are access types which refer to the "CAR" record and the "PERSON" record. The "PERSON" record has fields of both of these access types. Notice also that the types "LINK" and "DATE" are access types which are imported from other packages. Figure 1.14 shows the graphic depiction of this code.

This concludes the first section of this thesis. At this point, the reader should have acquired an understanding of the basic concepts of this visual language. In the following chapters a more formal description of this language will be given. In the process, it will also be proven that all necessary syntactic contingencies are addressed by the language.
Figure 1.1 Goguen's notation applied to the ADT of nat
Figure 1.2 Goguen's notation applied to the ADT of set
Table 1.1 Specification for package TEXT_HANDLER

package TEXT_HANDLER is

    type TEXT (MAXIMUM_LENGTH : INDEX) is limited private;

    function LENGTH (t : TEXT) return INDEX;
    function VALUE (t : TEXT) return STRING;
    function EMPTY (t : TEXT) return BOOLEAN;

    function TO_TEXT (S : STRING; max : INDEX) return TEXT;
    function TO_TEXT (C : CHARACTER; max : INDEX) return TEXT;
    function TO_TEXT (S : STRING) return TEXT;
    function TO_TEXT (C : CHARACTER) return TEXT;

    function "&" (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
    function "&" (LEFT : TEXT; RIGHT : STRING) return BOOLEAN;
    function "&" (LEFT : STRING; RIGHT : TEXT) return BOOLEAN;
    function "&" (LEFT : TEXT; RIGHT : CHARACTER) return BOOLEAN;
    function "&" (LEFT : CHARACTER; RIGHT : TEXT) return BOOLEAN;

    function "=" (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
    function "," (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
    function "+" (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
    function ">" (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;
    function ">=" (LEFT : TEXT; RIGHT : TEXT) return BOOLEAN;

    procedure SET (OBJECT : in out TEXT; VALUE : in TEXT);
    procedure SET (OBJECT : in out TEXT; VALUE : in STRING);
    procedure SET (OBJECT : in out TEXT; VALUE : in CHARACTER);

    procedure APPEND (TAIL : in TEXT; TO : in out TEXT);
    procedure APPEND (TAIL : in STRING; TO : in out TEXT);
    procedure APPEND (TAIL : in CHARACTER; TO : in out TEXT);

    procedure AMEND (OBJECT : in out TEXT; BY : in TEXT; POSITION : in INDEX);
    procedure AMEND (OBJECT : in out TEXT; BY : in STRING; POSITION : in INDEX);
    procedure AMEND (OBJECT : in out TEXT; BY : in CHARACTER; POSITION : in INDEX);

    function LOCATE (FRAGMENT : TEXT; WITHIN : TEXT) return INDEX;
    function LOCATE (FRAGMENT : STRING; WITHIN : TEXT) return INDEX;
    function LOCATE (FRAGMENT : CHARACTER; WITHIN : TEXT) return INDEX;
Figure 1.3  An attempt to use Goguen’s notation to describe the package of Table 1.1
Figure 1.4 First notation describing the package of Table 1.1
Figure 1.5 Key to first, second and third notations
Figure 1.7 Third notation describing the package of Table 1.1
<table>
<thead>
<tr>
<th>TEXT_HANDLER</th>
<th>TEXT</th>
<th>INDEX</th>
<th>STRING</th>
<th>CHARACTER</th>
<th>BOOLEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>VALUE</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>EMPTY</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&quot;=&quot;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&quot;&lt;&quot;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&quot;&lt;=&quot;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&quot;&gt;&quot;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>&quot;&gt;=&quot;</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>APPEND</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>APPEND</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>APPEND</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>AMEND</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>AMEND</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>AMEND</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>LOCATE</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>LOCATE</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
<tr>
<td>LOCATE</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td>Δ</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.8 Tabular notation describing the package of Table 1.1
Figure 1.9 Lower level of the second occurrence of function LOCATE in Table 1.1
Figure 1.10 Lower levels of the second and third occurrences of function "&" in Table 1.1
package ALBUMS is

    type TITLE is new STRING (1..40);

    type ARTIST is new STRING (1..40);

    type STYLE is (CLASSICAL, JAZZ, ROCK, COUNTRY,
                    SHOWS, RELIGIOUS, BALLROOM, PATRIOTIC,
                    FOREIGN, FOLK, BLUES, CHILDREN);

    type YEAR is range 1877..INTEGER'LAST;

    type LENGTH is delta 0.01 range 0.0..60.0;

    type NAME is new STRING (1..40);

    type SONG is
        record
            THE_NAME   : NAME;
            THE_LENGTH : LENGTH;
        end record;

    subtype NUMBER is POSITIVE range 1..30;

    type SONGS is array (NUMBER range <>) of SONG;

    type ALBUM (NUMBER_OF_SONGS : NUMBER := 10) is
        record
            THE_TITLE   : TITLE;
            THE_ARTIST  : ARTIST;
            THE_STYLE   : STYLE;
            THE_YEAR    : YEAR;
            THE_SONGS   : SONGS (1..NUMBER_OF_SONGS);
        end record;

end ALBUMS;
Figure 1.11 An attempt to use a tabular notation to describe the type dependencies of Table 1.2
Figure 1.12 Iconic notation describing the type dependencies of Table 1.2
Table 1.3 Specification for package LINKED_LIST

package LINKED_LIST is

    type CELL;

    type LINK is access CELL;

    type CELL is
        record
            VALUE : INTEGER;
            SUCC  : LINK;
            PRED  : LINK;
        end record;

end LINKED_LIST;
Figure 1.13  Iconic notation describing the package of Table 1.3
Table 1.4 Specification for package MUTUALLY_DEPENDENT

package MUTUALLY_DEPENDENT is

  type PERSON(SEX : GENDER);  
  type CAR;

  type PERSON_NAME is access PERSON;  
  type CAR_NAME is access CAR;

  type CAR is
    record
      NUMBER : INTEGER;  
      OWNER   : LINK;
    end record;

  type PERSON(SEX : GENDER) is
    record
      NAME      : INTEGER;  
      BIRTH     : DATE;  
      AGE       : INTEGER range 0 .. 130;  
      VEHICLE   : CAR_NAME;  
      case SEX is
        when M  => WIFE : PERSON_NAME(SEX => F);  
        when F  => HUSBAND : PERSON_NAME(SEX => M);
      end case;
    end record;

end MUTUALLY_DEPENDENT;
Figure 1.14 Iconic notation describing the package of Table 1.4
CHAPTER 2
ORGANIZATION OF THESIS VIA THE PACKAGE SPECIFICATION

2.1 Assumptions

We will now begin to formally describe the visual language. The Backus-Naur Form (BNF) grammar is used to organize the thesis. That is, the entire thesis has been organized via the Backus-Naur description of an Ada package specification.

The package specification is used for both historical and pragmatic reasons. The original intention of this thesis was to describe a visual language only for the Ada package specification. As work progressed, it became apparent that a visual representation must be designed for all of the program units since any program unit specification may be nested within a package specification. However, once this was realized, the package specification organization of the thesis was retained, since it provided a good way to demonstrate the completeness of the language.

Although some simplifying assumptions and liberties may have been made to ease the development and implementation of the language, such liberties will not prevent the developer from using all relevant features of the Ada language - such exceptions resulted mainly in stylistic changes.

Before we begin to discuss the language, it must be made clear that the intention of this paper is not to teach the reader the Ada programming language. At times the reader may be reminded of the purpose for some of Ada's features, and in some cases a brief
description may be given for those features which are seldom used or those which are particularly difficult to understand. However, the thesis is written with the assumption that the reader is already familiar with Ada.

It is also assumed that the reader has a working knowledge of Backus-Naur Form grammars.

Finally, the reader must be informed that a full traversal of all of the branches of the BNF rules will not be made. Only significant nonterminals will be expanded. The expansions of the less important nonterminals (e.g. identifier, type mark) are left to the reader. To expand such nonterminals would unnecessarily complicate the thesis. Once written, rewrite rules will be referenced, rather than rewriting the same nonterminal more than once.

2.2 Top-down Organization of Thesis

In this section, the reader can expect to learn the "big picture" and how the language's notations (some of which were previously described) interact to form an Ada package specification. But before continuing further, we must first study the BNF description of an Ada package specification. The highest level of the visual language will follow this description. The BNF description is as follows [Gehani84]:

```
package_specification ::=  
  package  identifier is  
    {basic_declarative_item}  
  [private
```
The identifier is simply the name imparted to the package and is equivalent to the package simple name at the end of the specification. We may envision that the graphical representation of the package specification will exist inside of a window, and that the package name will label the window, where it can be easily referenced. Other possibilities for the labeling of package specifications exist; it is the responsibility of the implementor to experiment with various interfaces.

As we can clearly see, a specification is usually divided into two parts: a visible portion and a hidden (private) portion (although in an extreme case, the visible portion may be empty; and a private part may not be included). In the graphic notations which we will later discuss, private declarations are denoted differently from other declarations. Instead of sequestering private types from other types, the visual notation will mark types as being private. This will remove the need for the partitioning of specifications, as well as eliminate the need to declare the type twice (once in the public portion, and once in the hidden portion).

We are left with the main part of this thesis, the basic declarative item. If it can be demonstrated that all of the syntactic contingencies of the basic declarative item can be incorporated into the visual language (and that the visual language is indeed worthwhile), then the efforts to develop this thesis have been
successful.

Now let us look at the BNF rule for the basic declarative item. It is as follows [Gehani84]:

\[
\text{basic\_declarative\_item ::= basic\_declaration | representation\_clause | use\_clause}
\]

Here we will make a slight break from the proposed method of organization. The discussion of the representation clause and the use clause will be delayed until we have discussed all of the components of the basic declaration. This is done because the basic declaration represents what most people think of as the "meat" of a specification. As such, the basic declaration will probably be the most interesting portion to the majority of readers. Here is the BNF grammar [Gehani84]:

\[
\text{basic\_declaration ::= object\_declaration | number\_declaration | type\_declaration | subtype\_declaration | subprogram\_declaration | package\_declaration | task\_declaration | generic\_declaration | exception\_declaration | generic\_declaration | renaming\_declaration | deferred\_constant\_declaration}
\]
One section will be devoted to each of these components. After all of the basic declaration components have been discussed, the representation clause and the use clause will each be discussed in their own sections.
CHAPTER 3
OBJECT DECLARATION

3.1 Simple Object Declarations

Before speaking about this syntactic class, I will show all of the BNF grammars which are relevant to the object_declaration. They are as follows from [Gehani84]:

\[
\text{object\_declaration} ::= \\
\quad \text{identifier\_list} : [\text{constant}] \text{ subtype\_indication} \\
\quad \quad ::= [\text{expression}] \\
\quad \quad \mid \text{identifier\_list} : [\text{constant}] \text{ constrained\_array\_definition} \\
\quad \quad \quad ::= [\text{expression}] \\
\]

\[
\text{subtype\_indication} ::= \\
\quad \text{type\_mark} [\text{constraint}] \\
\]

\[
\text{type\_mark} ::= \\
\quad \text{type\_name} \mid \text{subtype\_name} \\
\]

\[
\text{constraint} ::= \\
\quad \text{range\_constraint} \\
\quad \mid \text{floating\_point\_constraint} \\
\quad \mid \text{fixed\_point\_constraint} \\
\quad \mid \text{index\_constraint} \\
\quad \mid \text{discriminant\_constraint} \\
\]

\[
\text{range\_constraint} ::= \\
\quad \text{range} \ \text{range} \\
\]

\[
\text{range} ::= \\
\quad \text{range\_attribute} \\
\quad \mid \text{simple\_expression} .. \text{simple\_expression} \\
\]

\[
\text{floating\_point\_constraint} ::= \\
\]

39
floating_accuracy_definition [range_constraint]

floating_accuracy_definition ::= 
    digits static_simple_expression

fixed_point_constraint ::= 
    fixed_accuracy_definition [range_constraint]

fixed_accuracy_definition ::= 
    delta static_simple_expression

index_constraint ::= 
    (discrete_range {, discrete_range})

discrete_range ::= 
    discrete_subtype_indication | range

discriminant_constraint ::= 
    (discriminant_association {, discriminant_association})

discriminant_association ::= 
    [discriminant_simple_name {, discriminant_simple_name} =>] 
expression

constrained_array_definition ::= 
    array index_constraint of component_subtype_indication

Although there is a lot of information here, we will describe the visual representation for most of these rules. Many of the nonterminals above will be used in subsequent chapters, so we will refer to this section frequently.

There are really only two types of object declarations: constants and variables. An object is a constant if and only if the reserved word "constant" is used in its declaration. If an object is a
constant, it is usually given an initial value in its declaration. If the reserved word "constant" is not used, the object is a variable and may or may not be given an initial value in its declaration.

Objects will be represented by rectangular shapes that have rounded ends. Doubled borders will indicate constants, while single borders will indicate variables. The use of rectangular shapes (although not actually rectangles) is an allusion to the relationship between types (denoted by true rectangles) and objects (denoted as stated above). The distinction between constants and variables via bordering is used since a doubled oval may seem "thicker", "stronger" and hence less prone to change. Since constants do not take new values, perhaps this explanation will be useful.

At the highest level of abstraction, objects will be displayed and their dependencies on types will be denoted in the same manner that type dependencies were denoted in the introduction. Lower levels of abstraction will contain the initial values, and all constraints.

The initializing expression will be displayed in an oval. These ovals will be connected to the object via a doubled arc. The double arc is used since it seems to imply equality because of the equality symbol ("="). If an object is used in the expression of another object, there is no need to portray dependencies of expressions on objects via arcs. Such dependencies will be inferred by the code generator.

No matter which kind of object declaration is used,
constraints may be needed to describe the object. We must first state that all constraints will be placed in rectangular icons with thick, black borders. An undirected arc will connect the icon to the entity being constrained. In some cases, this entity will be the relationship between two other entities. For example, suppose that an object is declared to be a certain type, but the object declaration also uses a constraint. The constraint does not actually modify the type; the original type declaration remains unchanged. To connect the constraints directly to the type might mislead some users into thinking that the constraints are part of the original type declaration. To connect the constraints to the object is also misleading since the constraints are actually modifying the original type declaration. The constraint in question does, however, modify the object's dependency (or relationship) with the type.

For this reason, the undirected arc would be drawn from the constraint rectangle to the arc representing the object's dependency on the type. Showing a constraint as modifying the dependency seems to me to be somewhat akin to the way in which relationships can have attributes in the E-R model (used for database design). Of course, constraints which are actually a part of the original type declaration are of no concern to us in describing object declarations. An object with no constraints of its own, and based upon a type declaration described with the use of constraints, would simply be shown as depending upon that type; no constraints would be attached to the arc depicting the dependency.
Notations for the conventions described above are illustrated in Figure 3.1.

Now that we have discussed the placement of constraints in the object description, let us examine the descriptions for every type of constraint.

3.1.1 Constraints

The range constraint can consist of either the name of another range constraint, or two expressions which give the values of the range's boundaries. Range constraints which consist of the name of another range constraint will be denoted by placing the name of the range in the constraint icon. When depicting the other type of range constraint, both expressions will be represented individually. Expressions will be represented using oval icons. Hence, the range constraint will be shown as two such expressions placed left to right in ascending order. Between the two icons will be drawn an arc, directed in both directions. This "double direction" is an effort to denote the range, as if a number line were being shown. The entire structure is then placed into the appropriate constraint rectangle. The visual notation for range constraints are illustrated in Figure 3.2 and an example is given in Figure 3.3.

If a floating point constraint is used, it is necessary to declare how many digits of accuracy are needed. In Ada, this is accomplished through the use of the "digits" reserved word and a static simple expression. In our graphic language we want to denote such a constraint by placing a "digits" icon (designed to symbolize
the number of digits) in front of the static simple expression. The visual notation of a floating point constraint is given in Figure 3.4.a. An example of a floating point constraint is shown in Figure 3.5.a.

When specifying a fixed point constraint, Ada uses the reserved word "delta". To visually define a fixed point constraint, the greek character delta will be placed in front of the static simple expression. Again, both the delta character and the static simple expression will be contained inside the constraint icon. The notation for a fixed point constraint is given in Figure 3.4.b and an example is shown in 3.5.b.

Some declarations of objects of type real may require the use of a range constraint as well as the use of a floating point constraint or a fixed point constraint. This results in the use of multiple constraints for a single object declaration. These multiple constraints will simply be depicted as being "stacked" (see Figure 3.4 and 3.5).

The distinguishing feature of index constraints is the fact that one index constraint may contain multiple constraints. Simply stacking these constraints may result in confusion if a declaration makes use of an index constraints as well as other constraints. To indicate that all of the constraints are, in fact, portions of the one index constraint, they will all be listed inside of the same constraint rectangle. These constraints will be separated by thin, horizontal lines. If an index constraint consists of only one constraint, a thin line will still be used to distinguish this
constraint from other types of constraints. The visual notation of an index constraint is given in Figure 3.6. An example is given in 3.7.

Discriminant constraints are somewhat different from the other syntactic components which we have described. Discriminant constraints may only be used in subtype indications, and following a type mark. We will denote them in a manner which is similar to the method used to depict index constraints since, like index constraints, one discriminant constraint may be composed of many discriminant associations. Similarly, each discriminant association may contain many discriminant simple names. This is shown in Figure 3.8. Notice that, instead of using an undirected arc to denote the modification of the type dependency, we use a directed arc.

3.2 Objects of Constrained Array Type

The last syntactic component which must be considered is the constrained array definition. Objects which are defined as such will be shown as being dependent upon a nameless array type. The index constraint of the object declaration will be depicted as modifying the nameless array icon, rather than the dependency of the object on the array icon. If a type mark is used in the index constraint, the array icon will be shown to be dependent upon the corresponding component subtype indication. The notations for constrained array type declarations are given in Figure 3.9. Figure 3.10 depicts the object COLOR_TABLE (of constrained array type).
Figure 3.1.a Notation for the high level of a variable declaration

Figure 3.1.b Notation for the high level of a constant declaration

Figure 3.1.c Notation for the low level of a variable declaration

Figure 3.1.d Notation for the low level of a constant declaration
Figure 3.2.a Notation for a range constraint defined using a range identifier

Figure 3.2.b Notation for a range constraint defined using two simple expressions
SUMMER

Figure 3.3.a Example of range constraint which is defined using enumeration type SUMMER

JUNE AUGUST

Figure 3.3.b Example of range constraint which is defined using simple expression JUNE and simple expression AUGUST
Figure 3.4.a  Notation for a floating point constraint

Figure 3.4.b  Notation for a fixed point constraint
Figure 3.5.a  Example of a floating point constraint

Figure 3.5.b  Example of a fixed point constraint
Figure 3.6  Notation for an index constraint
Figure 3.7  Example of an index constraint
Figure 3.8.a Notation for the high level of a discriminant constraint applied to a subtype declaration

Figure 3.8.b Notation for the low level of a discriminant constraint
Figure 3.9.a Notation for the high level of a variable of constrained array type declaration

Figure 3.9.b Notation for the high level of a constant of constrained array type declaration

Figure 3.9.c Notation for the low level of a variable of constrained array type declaration

Figure 3.9.d Notation for the low level of a constant of constrained array type declaration
Figure 3.10.a Example of the high level of constrained array object COLOR_TABLE

Figure 3.10.b Example of the low level of constrained array object COLOR_TABLE from:

COLOR_TABLE : array(1 .. N) of COLOR;
CHAPTER 4
NUMBER DECLARATION

The BNF grammar for a number declaration is as follows [Gehani84]:

number_declaration :=
    identifier_list : constant := universal_static_expression;

A number declaration is actually a kind of object declaration (constant declaration). The difference is that number declarations do not need to specify their type. This is because these declarations use special universal types: universal integer type or universal real type. Which of these to use is implied by the initial value of the declaration. Since no types are explicitly used, none will be shown in the graphic notation. The use of a rounded rectangle-like icon is used to denote these constants, in the same fashion that we described the constant objects. As before, the values of these constants are accessible at lower levels. The graphic notation is shown in Figure 4.1. An example is shown in Figure 4.2.
Figure 4.1.a Icon for a number constant declaration

Figure 4.1.b Notation for the low level of a number constant
Figure 4.2.a  Example of the icon of the number constant $\pi$

Figure 4.2.b  Example of the low level of the number constant $\pi$
CHAPTER 5
TYPE DECLARATION

5.1.1 BNF and Organization of Type Declarations

As before, we will begin by looking at the BNF description [Gehani84].

type_declaration ::= 
  full_type_declaration 
  | incomplete_type_declaration 
  | private_type_declaration 

full_type_declaration ::= 
  type identifier [discriminant_part] is type_definition;

discriminant_part ::= 
  (discriminant Specification {; discriminant Specification})

discriminant specification ::= 
  identifier_list : type_mark [= expression]

type_definition ::= 
  enumeration_type_definition 
  | integer_type_definition 
  | real_type_definition 
  | array_type_definition 
  | record_type_definition 
  | access_type_definition 
  | derived_type_definition

Instead of expanding all of the nonterminals now, we will devote the sections of this chapter to the rewriting and explanation of those nonterminals which are considered to be particularly
important. We will begin by expanding all of the syntactic components of the full type declaration and showing their corresponding graphic representations. After doing so, we will describe the representations for incomplete type declaration and private type declaration, respectively. The full type declaration represents a very large part of the package specification. For this reason, the components of the full type declaration are elevated one level in the organization of the grammar (placed at the same level of incomplete type declaration and the private type declaration).

5.1.2 Discriminant Part

Let us begin our description of the full type declaration by examining the discriminant part.

A discriminant is simply a type of parameter for a type. Discriminants are used in record type declarations. They provide the ability to pass items, such as the boundary values of range constraints, to the type declaration. Discriminants often require default values to be set in case no such value is passed.

To denote the existence of a discriminant, we will use a small, empty square. Since discriminants are too complex to develop a visual notation for all occurrences, they must be entered at the editor level. The discriminant section of code will be accessed by clicking the mouse button while pointing to the discriminant square (mentioned above). The notation is shown in Figure 5.1.

5.2 Type Definition
Now let us expand the various syntactic components of the type definition. We will expand each component in turn, and show the corresponding graphic notation for the component. We will describe the visual representation by associating it with the BNF grammar. We will show both the high levels of abstraction as well as the lower levels. We will also include examples to demonstrate how the notation may be used.

5.2.1 Enumeration Type Definition

We will start with the enumeration type definition as given by [Gehani84]:

```
enumeration_type_definition ::= (enumeration_literal_specification
                            {,enumeration_literal_specification})
enumeration_literal_specification ::= enumeration_literal

enumeration_literal ::= identifier | character_literal
```

Enumeration types will be denoted via a special rectangular icon. The small marks at the bottom of this rectangle are an effort to symbolize the nature of enumeration types (i.e. composed of discrete elements in a specific order). The enumeration literals which make up the enumeration will be accessible at a lower level of abstraction (by using the pointing device). The graphic notation is given in Figure 5.2. Examples of the language features for
enumerations are given in Figure 5.3.

5.2.2 Integer Type Definition

Now let us look at the next type definition syntactic unit. It is the integer type definition [Gehani84].

\[
\text{integer\_type\_definition ::= range\_constraint;}
\]

An ordinary rectangle will be used to denote these and other simple types (see Figure 5.4). Since the type integer is inferred, the dependency on the type of integer will not be demonstrated in the type dependency graphs. As before, constraints will be accessed via the pointing device. An example of an integer type declaration is given in Figure 5.5.

5.2.3 Real Type Definition

Next we examine the real type definition [Gehani84].

\[
\text{real\_type\_definition ::=}
\begin{align*}
\quad & \text{floating\_point\_constraint} \\
\quad & | \text{fixed\_point\_constraint}
\end{align*}
\]

From inspecting the syntax, it is obvious that we need a way to denote that a declaration is of the type real. We also may need a method for distinguishing whether or not an object is a floating-point or fixed-point type. Should this distinction be made at a higher or a lower level? Although the two are not compatible,
they are so similar in essence that distinctions between the two will be shown at a lower level of abstraction. Recall that the definitions of fixed point constraint and floating point constraint have been previously given in the section on object declarations. The plain rectangle icon for simple types (used by the integer type definition) is used for real type definitions. Like the integer definition, the reliance upon the type of real is implicit in the declaration and allows us to ignore depicting the use of a type dependency arc. Notations for each form of the real type definition may be viewed in Figure 5.6. Examples can be seen in Figure 5.7.

5.2.4 Array Type Definition

The next syntactic component that we will examine is the array type definition [Gehani84].

array_type_definition ::=  
  unconstrained_array_definition  
  | constrained_array_definition

unconstrained_array_definition ::=  
  array(index_subtype_definition {, index_subtype_definition})  
  of component_subtype_indication

index_subtype_definition ::=  
  type_mark range <>

All arrays are represented at the highest level with a special rectangular icon which is an attempt to symbolize arrays. Since I feel that the essences of constrained and unconstrained arrays are
the same, I will not make distinctions between them at the highest level of abstraction. Instead, these differences can be implied by the index definitions at a lower level. Constrained array declarations are made without a type mark. The representation of constrained arrays, shown in Chapter 3, will not be repeated here. Figure 5.8 will show the notation of an unconstrained array type declaration, while Figure 5.9 gives an example of such a declaration.

Type string is a predefined array type. It is not inferred by the compiler in type declarations, so we will include type string when we demonstrate type dependencies. Types which are based upon the string type will be shown as arrays.

5.2.5 Record Type Definition

Now we will continue by describing the record type definition. The grammar is as follows [Gehani84]:

```
record_type_definition ::= 
  record
    component_list
  end_record

component_list ::= 
  component_declaration (component_declaration)
  | {component_declaration} variant_part
  | null;

component_declaration ::= 
  identifier_list : component_subtype_definition [= expression];

component_subtype_definition ::= 
  subtype_indication
```
variant_part ::= 
    case discriminant_simple_name is 
        variant 
        { variant } 
    end case; 

variant ::= 
    when choice { | choice} => 
        component_list 

choice ::= 
    simple_expression 
    | discrete_range 
    | others 
    | component_simple_name 

The record type is built from other type declarations (those we have already discussed and those which we will soon discuss). The record type, like all of the other types we have discussed, is shown to be dependent on other types by using directed arcs. When using this convention, the question arises of how to handle the situation in which a record has more than one field of a given type. The answer is simply that, at the highest level, one and only one arc is drawn from that type to the record. The arcs at the highest level show the dependence of one type on the existence of another. How many fields refer to the type is a detail which is relevant only at a lower level of abstraction. At a lower level, it is necessary to represent every component individually. This is necessitated by the fact that each field may contain a unique initializing expression.

The expanded view will show all component declarations which
includes component identifiers and initializing expressions. The identifiers will be placed upon the arc from the type to the record icon. The initializing expressions will be drawn in the same manner as initializing expression for objects, but will be connected to the identifiers (see Figure 5.10). An example of a record type declaration is given in Figure 5.11.

Variants provide record types the ability to specify alternate component lists. The structure of such a record type is determined by the passing of values to determinants.

When variants are used, the record type declaration depends on the definition of all types referenced within the variant part, although, in practice, some of these may never be used. This dependency should be shown at the highest level of abstraction. At the highest level of abstraction the variant icon will be used (the variable icon rotated 90 degrees, containing a question mark). The types used by the variables will be shown via type dependency arcs. As above, each type will be represented once and only once.

Once the record type is expanded, the variant part will reveal its lower level. The discriminant simple name will be shown to modify the variant icon. The variants in the case statement will show their choice components and corresponding components list. Each component list will be shown as being related to a group of choices. These choices will be represented using a new icon. The icon is a circle which, like the variant icon, contains a question mark. The component list will be similar to the highest level of a
record declaration without variant parts; directed arcs will be drawn from the component declaration subtype indications to the choice icons. The notation is shown in Figure 5.12 and an example is given in Figure 5.13.

The component lists of the variant part may, in turn, be expanded. Thus, expansions may continue for many levels (although hopefully the nesting of variants will be somewhat limited).

5.2.6 Access Type Definition

Now let us discuss the access type definitions [Gehani84].

\[
\text{access\_type\_indication} ::= \\
\text{access}\ \text{subtype\_indication}
\]

Access types are what are referred to in most programming languages as pointers. It is for this reason that the graphic notation of a large arrow has been adopted. Recall that since access types point to an instance of a specific type, an arc is drawn from the referenced type declaration to the access type, rather than the arc indicating the "pointing action" of the access type. In a moment, we will discuss the handling of the incomplete type declarations; a topic which is a direct result of the use of access types. Figure 5.14 depicts the graphic notation. Figure 5.15 presents an example.

5.2.7 Derived Type Definition

The derived type definition is the last of the type definition components that we must describe. The BNF from [Gehani84]:

derived_type_definition ::= 
   new subtype_indication

The derived type is formed with two icons of the type which the derived type is based upon. One of the icons will be superposed on the other, to denote that the derived type is based upon another type declaration. If a record type is to be derived from a previous record declaration, the new type is shown as a record icon, with another record icon superposed upon it (see Figure 5.16). The new type is shown as being dependent only upon the base type. Thus, the nature of the new type is not lost as types are derived from other types. Yet, this method has the benefit that it is unnecessary to duplicate the type dependencies of the "base type". The dependencies can be thought of as being inherited. The notation is presented in Figure 5.17 and an example in Figure 5.18.

5.3.1 Incomplete Type Declaration

Now I will return to the two items which I passed over earlier: the incomplete type declaration and the private type declaration. Let us first examine the incomplete type declaration. From [Gehani84]:

incomplete_type_declaration ::= 
   type identifier [discriminant_part];

Incomplete type declarations are used for defining recursive
types. Each incomplete type declaration is associated with a full type declaration. Also, the discriminant part of the incomplete declaration is used if and only if it is used in the full declaration. The incomplete declaration is motivated by the arrangement of declarations in the specification text. For this reason, I maintain that situations in which incomplete declarations are needed can be automatically detected. Thus, the visual language should be able to automatically introduce an incomplete declaration into the code when it is necessary to do so. Therefore, there is no need for a graphic depiction of such an incomplete type declaration.

5.3.2 Private Type Declaration

Finally, we close this section by describing the visual translation of a private type declaration. From [Gehani84]:

```
private_type_declaration ::= 
type identifier [discriminant_part] is [limited] private;
```

Private types support the principle of data abstraction. The use of private types ensures that other packages will not improperly access instances of the type. The use of the "limited" option puts additional safeguards on the type by denying the implicit operations (i.e. assignment, equality and inequality) to be used by other packages.

Types which are private will still retain the basic icons of the visible types. The difference in visibility will be shown by drawing
the icons with dashed lines to suggest their hidden nature. Since unlimited private types can be thought of as being less hidden, the dashes which comprise the symbols of such types (see Figure 5.19) will be larger than those which comprise the symbols of limited private types (see Figure 5.20). The latter will have a dotted outline because of their less visible nature.
Figure 5.1.a Notation for the high level of a declared type with a discriminant

Figure 5.1.b Notation for the low level of a declared type with a discriminant
Figure 5.2.a  Icon for an enumeration type declaration

Figure 5.2.b  Notation for the low level of an enumeration type declaration
Figure 5.3.a  Example of the icon of enumeration type COLORS

Figure 5.3.b  Example of the low level of the enumeration type COLORS from:

```
type COLORS is (RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET);
```
Figure 5.4.a Icon for an integer type declaration

Figure 5.4.b Notation for the low level of an integer type declaration
Figure 5.5.a  Example of the icon of the integer type PAGE_NUM

Figure 5.5.b  Example of the low level of the integer type PAGE_NUM from:

    type PAGE_NUM is range 1 .. 200;
Figure 5.6.a Icon for a real type declaration

Figure 5.6.b Notation for the low level of a floating point type declaration

Figure 5.6.c Notation for the low level of a fixed point type declaration
Figure 5.7.a Example of the low level of floating point type MASS from:

```
type MASS is digits 5 range 0.0 .. 1.0;
```

Figure 5.7.b Example of the low level of fixed point type VOLTAGE from:

```
type VOLTAGE is delta 0.1 range -12.0 .. 24.0;
```
Figure 5.8.a  Icon for an array type declaration

Figure 5.8.b  Notation for the high level of an unconstrained array type declaration

Figure 5.8.c  Notation for the low level of an unconstrained array type declaration
Figure 5.9.a  Example of the icon of unconstrained array type VECTOR

Figure 5.9.b  Example of the high level of unconstrained array type VECTOR

Figure 5.9.c  Example of the low level of unconstrained array type VECTOR from:

type VECTOR is array(INTEGER range <>) of REAL;
Figure 5.10.a Icon for a record type declaration

Figure 5.10.b Notation for a record type declaration with a component list of null

Figure 5.10.c Notation for the high level of a record type declaration without a variant part

Figure 5.10.d Notation for the low level of a record type declaration without a variant part
Figure 5.11.a Example of the high level of the record type DATE

Figure 5.11.b Example of the low level of the record type DATE from:

type DATE is
record
  DAY : INTEGER range 1 .. 31;
  MONTH : MONTH_NAME;
  YEAR : INTEGER range 0 .. 4000;
end record;
Figure 5.12.a Notation for the high level of a record type declaration with a variant part

Figure 5.12.b Notation for the low level of a record type declaration with a variant part
Figure 5.13.a  Example of the high level of the record type PERIPHERAL (which has a variant part)
Figure 5.13.b Example of the low level of the record type PERIPHERAL (which has a variant part) from:

```pascal
type PERIPHERAL(UNIT : DEVICE := DISK) is
  record
    STATUS : STATE;
    case UNIT is
      when PRINTER =>
        LINE_COUNT : INTEGER range 1 .. PAGE_SIZE;
      when others =>
        CYLINDER : CYLINDER_INDEX;
        TRACK : TRACK_NUMBER;
    end case;
  end record;
```
Figure 5.14.a Icon for an access type declaration

Figure 5.14.b Notation for the low level of an access type declaration
Figure 5.15.a  Example of the icon of the access type
STUDENT_POINTER

Figure 5.15.b  Example of the low level of the access type
STUDENT_POINTER
Figure 5.16  The set of icons denoting derived type declarations
Figure 5.17.a Icon for a derived type declaration

Figure 5.17.b Notation for the low level of a derived type declaration
Figure 5.18.a  Example of the icon of the derived type FAHRENHEIT

Figure 5.18.b  Example of the low level of the derived type FAHRENHEIT
Figure 5.19 The set of icons denoting private type declarations
Figure 5.20  The set of icons denoting limited private type declarations
CHAPTER 6
SUBTYPE DECLARATION

This section is concerned with the declaration of subtypes. The BNF grammar from [Gehani84] is given below.

```
subtype_declaration ::=  
    subtype identifier is subtype_indication;
```

It seems that almost every syntactic unit in this grammar has previously been discussed. We need only the visual representation of a subtype. A subtype symbol is composed of the icon of the subtype indication and two large arrowheads. The arrowheads are placed on both sides of the icon and are pointed towards each other. This is an attempt to symbolize the lesser nature of subtypes: their subrange nature. The various subtype symbols are shown in Figure 6.1. The formal visual notation is shown in Figure 6.2. An example of a subtype declaration is shown in Figure 6.3.
Figure 6.1 Various symbols for subtypes
Figure 6.2.a  Symbol for the high level of a subtype declaration

Figure 6.2.b  Notation for the low level of a subtype declaration
Figure 6.3.a  Example of the icon of the subtype SIZE

Figure 6.3.b  Example of the low level of the subtype SIZE
CHAPTER 7
SUBPROGRAM DECLARATION

7.1 BNF and Organization
This section is, perhaps, the most crucial to this thesis. It
was the idea of using Gouguen's methods to specify the declaration
of subprograms which was the inspiration of the entire thesis. Also,
it has been in the specification of the declaration of subprograms
where the most fruitful work has occurred.

Let us examine the grammar from [Gehani84].

subprogram_declaration ::= 
    subprogram_specification;

subprogram_specification ::= 
    procedure identifier [formal_part] 
    | function designator [formal_part] return type_mark

designator ::= 
    identifier | operator_symbol

operator_symbol ::= 
    string_literal

formal_part ::= 
    (parameter_specification {; parameter_specification})

parameter_specification ::= 
    identifier_list : mode type_mark [:= expression]

mode ::= 
    [in] | in out | out

96
The above syntax was completely described in the introductory section. However, for the sake of completeness it will be described again, in a more formal manner. In doing so, we will only refer to the table representation for subprogram declarations. The other notations in the introduction were shown for background purposes only.

7.2.1 High Level of Subprogram Specifications

As stated in the introduction, parameters are denoted by the placement of an arrowhead in the appropriate cell of the table. By appropriate we mean the cell which is located at the intersection of the row of the function in question and the column of the type of the parameter. The kind of arrowhead placed is very significant. An empty (white) arrowhead pointed toward the function name (right) denotes an "in" parameter. An empty arrowhead pointed away from the function name (left) indicates an "out" parameter. A parallelogram formed from two of the above empty arrowheads pointing away from each other denotes an "in out" parameter. Finally a full (black) arrowhead which points away from the function name denotes the value returned from a function.

The use of a black arrowhead is necessary and sufficient to denote that a subprogram is a function.

Notations for the high level of subprogram specifications are given in Figure 7.1. Examples of the high level of subprogram specifications are given in Figure 7.2.
7.2.2 Low Level of Subprogram Specifications

As demonstrated before, the ordering and naming of parameterers can be viewed at a lower level. By using the pointing device, the names of formal parameters and their true ordering within the subprogram specification can be learned. Default expressions may also be viewed at the lower level.

Notations for the low level of subprogram specifications are given in Figure 7.3. Examples of the low level of subprogram specifications are given in Figure 7.4.

7.3 Operations Performable on the Subprogram Table

The table notation can be enhanced by including helpful features. Some of these were alluded to previously. I will briefly describe the essential features here. The number of such operations is only limited by the imagination of future implementors.

The most important feature which we must discuss is that of joining single subprogram specifications to form large tables of subprograms. This is straightforward. Those subprogram specifications which are to be joined are marked. Next, the join operation automatically creates a template with all type marks of both specifications. The names of the subprograms and the appropriate mode indicators are automatically placed in the correct cells (see Figure 7.5). The lower level of each subprogram is accessed, separately, in the same manner described previously.

Next, we discuss ways of shrinking the table to facilitate easier viewing. This is done in two ways: yanking rows and yanking
columns.

In both of these methods a group of rows (or columns) is chosen. This group is then hidden from the view of the table (however the group has not actually been removed from the table). This is demonstrated in Figure 7.6 and Figure 7.7 which rank rows and columns, respectively, from the table of Figure 1.8.

Finally, we may envision a feature which condenses overloaded groups into single rows. Shading is used to denote that these single rows are indeed the condensation of overloaded groups. Also, no parameter modes are displayed in these rows to prevent users from referencing a subprogram without having viewed all of its instances. This operation is illustrated in Figure 7.8.
<table>
<thead>
<tr>
<th>identifier</th>
<th>type_mark</th>
<th>type_mark</th>
<th>(\ldots)</th>
<th>type_mark</th>
</tr>
</thead>
</table>

**Figure 7.1.a** Notation for the high level of a procedure with no parameters

<table>
<thead>
<tr>
<th>identifier</th>
<th>(&lt;)</th>
<th>(\lhd)</th>
<th>(\ldots)</th>
<th>(\lhd)</th>
</tr>
</thead>
</table>

**Figure 7.1.b** Notation for the high level of a procedure with assorted parameters

<table>
<thead>
<tr>
<th>designator</th>
<th>type_mark</th>
<th>type_mark</th>
<th>(\ldots)</th>
<th>type_mark</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\ldots)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.1.c** Notation for the high level of a function with no parameters

<table>
<thead>
<tr>
<th>designator</th>
<th>type_mark</th>
<th>type_mark</th>
<th>(\ldots)</th>
<th>type_mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\llangle)</td>
<td>(\llangle)</td>
<td>(\llangle)</td>
<td>(\ldots)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.1.d** Notation for the high level of a function with assorted parameters
Figure 7.2.a  Example of the high level of a procedure with no parameters

Figure 7.2.b  Example of the high level of a procedure with assorted parameters

Figure 7.2.c  Example of the high level of a function with no parameters

Figure 7.2.d  Example of the high level of a function with parameters
Figure 7.3.a Notation for the low level of a procedure

Figure 7.3.b Notation for the low level of a function
Figure 7.4.a Example of the low level of a procedure with no parameters

Figure 7.4.b Example of the low level of a procedure with parameters

Figure 7.4.c Example of the low level of a function with no parameters

Figure 7.4.d Example of the low level of a function with parameters
Figure 7.5.a The two subprograms SORT_LIST and EQUAL are marked to be joined.

Figure 7.5.b The result of joining the subprograms of Figure 7.5.a.
<table>
<thead>
<tr>
<th>TEXT_HANDLER</th>
<th>TEXT</th>
<th>INDEX</th>
<th>STRING</th>
<th>CHARACTER</th>
<th>BOOLEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td></td>
<td></td>
<td>▲</td>
<td></td>
<td>▲</td>
</tr>
<tr>
<td>VALUE</td>
<td></td>
<td></td>
<td>▲</td>
<td></td>
<td>▲</td>
</tr>
<tr>
<td>EMPTY</td>
<td></td>
<td></td>
<td>▲</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td></td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td></td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td></td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td></td>
</tr>
<tr>
<td>AMEND</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>AMEND</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>AMEND</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
</tr>
<tr>
<td>LOCATE</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td></td>
</tr>
<tr>
<td>LOCATE</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td></td>
</tr>
<tr>
<td>LOCATE</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.6 The result of yanking rows from the table in Figure 1.8
<table>
<thead>
<tr>
<th>TEXT_HANDLER</th>
<th>TEXT</th>
<th>INDEX</th>
<th>BOOLEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>VALUE</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>EMPTY</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>TO_TEXT</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>AMEND</td>
<td>△ △</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>AMEND</td>
<td>△ △</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>AMEND</td>
<td>△ △</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>LOCATE</td>
<td>△ △</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>LOCATE</td>
<td>△ △</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>LOCATE</td>
<td>△ △</td>
<td>△</td>
<td>△</td>
</tr>
</tbody>
</table>

Figure 7.7 The result of yanking columns from the table in Figure 7.6
Figure 7.8  The result of condensing overloaded subprograms in the table of Figure 7.7
CHAPTER 8
PACKAGE DECLARATION

This is the "recursive call" of this thesis. This thesis is organized as an explanation of how to represent a package declaration (see below) using a graphic notation. Since Ada package specifications can be nested, it is necessary to facilitate such nesting elegantly. Let us first look at the grammar for a package declaration [Gehani84].

```
package_declaration ::= 
    package_specification;
```

It should be clear that one cannot simply nest visual representations in the same way that textual representations can be nested. Because of this visual language's loose format, such attempts would be disastrous. In such a nesting, it would be nearly impossible to determine the ownership of declarations, objects, etc.

It is obviously necessary to clearly delineate nested packages from their containing packages. This will be accomplished by displaying the nested package as a symbol. The details of the package can be viewed by "zooming in" on the symbol. Once expanded, only those entities of the outer package which are used by the nested package are inaccessible. Of course, the nested package's lower level is represented using the notations and methods which are described in this and other sections of this thesis.
For the iconic representation of a package, I have chosen to use the icon which Booch uses in his classic book on Ada [Booch87]. Besides using this icon to represent nested packages, it may also be used packages to represent those which are not nested. That is, we can represent all packages, at their highest possible level, with such an icon. The package icon is presented in Figure 8.1.

Thus, a level, higher than any other level previously mentioned in this thesis, has just been described. A high level "library" of such icons can now be envisioned. A user who desires to reference an Ada package specification will browse through such a graphic library until finding the needed package specification. When the icon is expanded, the programmer is then able to see the grid notation shown in a previous section.

It is also possible that such a library might portray the interaction of Ada packages in a way similar to the manner in which type dependencies are denoted. Such a feature might be very helpful to programmers using the visual language. A detailed description of such additions will not be given here. These features are mentioned here to provide guidance to future implementors.
Figure 8.1 Icon for a package specification
CHAPTER 9
TASK DECLARATION

9.1 BNF and Basic Notation

Tasks provide the ability to use parallel processing. The BNF follows [Gehani84]:

task_declaration ::= 
  task_specification;

task_specification ::= 
  task [type] identifier [is
  {entry_declaration}
  {representation_clause}
  end [task_simple_name]]

entry_declaration ::= 
  entry identifier [(discrete_range)] [formal_part];

In depicting the highest level of a task specification, we will again turn to Booch's notation. An example of the representation of an Ada task specification can be seen in Figure 9.1.

9.2 Task Types

The first task feature we will discuss is the declaration of a task type. If the reserved word "type" is used in the task declaration, the result is not a simple task, but a task type which can be used to declare task objects. I will not concern myself with explaining task objects, the interested reader can refer to the Ada Reference Manual [Gehani84]. It is only necessary to state that task
types are limited private. As such, a task type will be denoted using the same convention that was developed to describe other limited private types: The task icon will be used to denote a task type, but the identifier portion will be drawn using the dotted manner which symbolizes the hidden nature of limited private types (see Figure 9.2).

The dependence of task objects upon task types will be denoted with a directed arc in a manner identical to the representation of other type dependencies. The task objects declarations will be represented with the task icon shown above (see Figure 9.1).

9.3 Entry Declarations

Next, it should be noticed that entry declarations and subprogram declarations are quite similar. For this reason, we will modify the tabular subprogram declaration notation to represent task entry declarations.

The primary modification is that we adopt a slanted table which is similar to our task icon. At the lower level of an entry specification, a discrete range is shown to be connected to the entry identifier, if such a range exists.

Also all of the viewing and accessing features, described for subprograms in Chapter 7, may be used to access task tables.

The visual notation for both the high level and the low level of an entry declaration is shown in Figure 9.3. An example of an entry declaration is given in Figure 9.4.

The representation clause will not concern us now. It will be
discussed in Chapter 15.
Figure 9.1 Icon for a task declaration
Figure 9.2 Icon for a task type declaration
Figure 9.3.a Notation for the high level of an entry declaration

Figure 9.3.b Notation for the low level of an entry declaration
Figure 9.4.a  Example of the high level of entry declaration READ

Figure 9.4.b  Example of the low level of entry declaration READ from:

entry READ (N : in INDEX; V : out ITEM);
CHAPTER 10

GENERIC DECLARATION

Generics allow types to be passed to them in a manner similar to the way that values and addresses are passed to formal parameters. Generics allow programmers to use the same algorithm on different data structures without replicating code. From [Gehani84]:

\[
\text{generic
declar\aion ::=}
\]
\[
\text{generic\specification;}
\]

\[
\text{generic\specification ::=}
\]
\[
\text{generic\formal\part subprogram\specification}
\]
\[
\text{generic\formal\part package\specification}
\]

\[
\text{generic\formal\part ::=}
\]
\[
\text{generic \{generic\parameter\declaration\}}
\]

\[
\text{generic\parameter\declaration ::=}
\]
\[
\text{identifier\list : [in [out]] type\mark [:= expression];}
\]
\[
\text{type identifier is generic\type\definition;}
\]
\[
\text{private\type\declaration}
\]
\[
\text{with subprogram\specification [is name];}
\]
\[
\text{with subprogram\specification [is <>];}
\]

\[
\text{generic\type\definition ::=}
\]
\[
\text{(<>)}
\]
\[
\text{range <>}
\]
\[
\text{digits <>}
\]
\[
\text{delta <>}
\]
\[
\text{array\type\definition}
\]
\[
\text{access\type\definition}
\]
To denote generics at a high level we will borrow a little, as before, from Booch's notation [Booch87]. However, we will not borrow all of his conventions. Generic packages will be represented with the same icon that was used to represent nongeneric packages. However, one of the conventions which we have developed so far will be utilized. Generic packages and subprograms will be shown as having a single border (see Figure 10.1). This is related to the way in which variables were shown to have single borders in contrast to constants. Although generic units are not variables, their whole purpose is to provide variety via instantiation. By viewing the first couple of rules for generics, it may be observed that all of the notations we have developed for subprograms and packages will be useful.

A very unique feature of generic unit specifications is the use of a generic formal part. Visually, we will represent these generic formal declarations as modifying the generic unit being defined. So a directed arc will be drawn from the formal declaration icon to the generic unit icon.

There are several types of generic parameter declarations. The first is the generic formal object.

Generic formal objects are the same as the variables and constants referred to earlier. These objects will be represented using the same symbols described previously. It is important to mention that these objects can have an optional mode of "in" or "in
out" the appropriate marker will be placed directly on the directed arc.

The second type of parameter declaration is the generic formal type. Such a declaration may take the form of a full type declaration or a private type declaration. As above, the notations already developed will be used when representing these parameters. The influence of a generic type definition will be shown in the same manner that a constraint is denoted.

The third and final type of parameter declaration is the generic formal subprogram. As the name suggests, this parameter is the declaration of a subprogram template. And again, the table notation for subprograms will be utilized. The optional "is" clause will be placed in a constraint box and will be shown as modifying the arc. The is clause will be placed verbatim, unless it is an array type definition or an access type definition (in which case, notations previously described will be used). The visual notation is given in Figure 10.2.
Figure 10.1.a  Icon for a generic package declaration

<table>
<thead>
<tr>
<th>identifier</th>
<th>type_mark</th>
<th>type_mark</th>
<th>...</th>
<th>type_mark</th>
</tr>
</thead>
</table>

Figure 10.1.b  Notation for the high level of a generic subprogram declaration
Figure 10.2.a Notation for the high level of a generic package and its generic parameters

Figure 10.2.b Notation for the low level of a generic package and its generic parameters
CHAPTER 11
EXCEPTION DECLARATION

This section describes the representation for exception declarations [Gehani84].

\[
\text{exception\_declaration ::= identifier\_list : exception;}
\]

It is obvious that this is a very simple declaration (the easiest that we will deal with). To denote the exception, an octagonal icon is used (this icon was intentionally chosen to because of its similarity to a stop sign. The notation is shown in Figure 11.1.a, and examples are shown in Figure 11.1.b.
Figure 11.1.a Icon for an exception declaration

Figure 11.1.b Example of the icon of exception OVERFLOW
CHAPTER 12
GENERIC INSTANTIATION

Generic unit declarations define a template for generic use. The generic unit declaration is almost like a type declaration in the sense that such a declaration is useless by itself. In order to make use of a type, objects have to be declared of that type. In order to make use of a generic unit, generic instantiations must be used to declare program units which are capable of real processing.

The BNF grammar from [Gehani84]:

generic_instantiation ::=  
  package identifier is  
    new generic_package_name [generic_actual_part];  
  | procedure identifier is  
    new generic_procedure_name [generic_actual_part];  
  | function designator is  
    new generic_function_name [generic_actual_part];

generic_actual_part ::=  
  (generic_association {, generic_association})

generic_association ::=  
  [generic_formal_parameter =>] generic_actual_parameter

generic_formal_parameter ::=  
  parameter_simple_name  
  | operator_symbol

generic_actual_parameter ::=  
  expression  
  | variable_name  
  | subprogram_name

125
Like a derived type which is created from another type, generic instantiations will be shown as descending from their corresponding generic declaration. This will be denoted with a directed arc from the declarations to the instantiation.

This directed arc notation is almost sufficient to represent generic instantiations. It is necessary, however, to mention one additional syntactic feature: Generic associations. Generic associations will be denoted using the same structure as index constraints. The code of each generic association will be placed verbatim in a separate cell of the structure. The notation for generic package instantiation and generic subprogram instantiation is given in Figure 12.1 and 12.2, respectively.
Figure 12.1.a  Notation for the high level of the instantiation of a generic package

Figure 12.1.b  Notation for the low level of the instantiation of a generic package
Figure 12.2.a Notation for the high level of the instantiation of a generic subprogram

Figure 12.2.b Notation for the low level of the instantiation of a generic subprogram
CHAPTER 13
RENAMEING DEClARATION

Renaming declarations allows a programmer to declare a new name for the specified entity. The BNF grammar from [Gehani84] follows:

```
renaming_declaration ::= 
  identifier : type_mark
  renames object_name;
  | identifier : exception
  renames exception_name;
  | package identifier renames package_name;
  | subprogram_specification
  renames subprogram_or_entry_name
```

A new icon will be drawn (the same type of icon as the original entity). To denote that the new icon represents a renaming, a double line will be drawn from the original entity to the renaming entity. The double line will begin at the border of the original, renamed object and end in an arrowhead, pointing to the new, renaming object (see Figure 13.1).

If a subprogram is being renamed, the line is drawn from either the left or right side of the subprogram's table, whether or not the subprogram specification shares a table with those of other subprograms.
Figure 13.1.a Notation for the renaming of an object

Figure 13.1.c Notation for the renaming of a package

Figure 13.1.d Notation for the renaming of a subprogram
CHAPTER 14
DEFERRED CONSTANT DECLARATION

Deferred constant declarations are related to private types. In the deferred constant declaration the full description of the constant is postponed to facilitate the hiding of the constant. The BNF grammar for deferred constant declarations is [Gehani84]:

\[
deferred\_constant\_declaration ::= \text{identifier\_list : constant type\_mark;}
\]

When declaring a private constant in Ada, two steps are necessary. The first is to write a deferred constant declaration. The second step is to accompany this partial declaration with a full declaration in the private portion of the package specification.

What was done for private declarations will now be done for deferred constant declarations. That is, we will make it unnecessary to use two declarations for the same constant. Our notation to represent deferred constants simply consist of merging the notation for visible constants with the notation for private declarations: dashed icons to represent ordinary private constants, dotted icons to represent limited private constants (see Figure 14.1).
Figure 14.1.a  Icon for a private constant

Figure 14.1.b  Icon for a limited private constant
CHAPTER 15
REPRESENTATION CLAUSE

The representation clause is used as a method of specifying implementation-dependent procedures. The syntax is as follows [Gehani84]:

representation_clause ::= 
  type_representation_clause
  | address_clause

type_representation_clause ::= 
  length_clause
  | enumeration_representation_clause
  | record_representation_clause

length_clause ::= 
  for attribute use simple_expression;

enumeration_representation_clause ::= 
  for type_simple_name use aggregate;

aggregate ::= 
  {component_association {, component_association}}

component_association ::= 
  [choice {, choice} => ] expression

record_representation_clause ::= 
  for type_simple_name use
    record [alignment_clause]
      {component_clause}
    end record;

alignment_clause ::=
at mod static_simple_expression;

component_clause ::=  
    component_name at static_simple_expression range static_range;

address_clause ::=  
    for simple_name use at simple_expression;

Since the real purpose of representation clauses is to specify the implementation of certain constructs and program entities, their use does not affect any of the logical structure of an Ada program. Because of their low-level, implementation-dependent nature, no visual representation will be given to these clauses. If a programmer cares to place such clauses in code, or a user of the language wishes to view such clauses in code, he must do so via the editor which is accessed by "dropping through" the lowest visual level.
CHAPTER 16
USE CLAUSE

The use clause specifies the utilization of a package, making all of the entities of that package visible inside of the package which contains the use clause. The syntax is [Gehani84]:

```
use_clause ::= 
    use package_name {, package_name};
```

The use clause will simply be denoted by a package icon enclosed in a rectangle. This rectangle will distinguish the package from one which is nested (see Figure 16.1).
Figure 16.1  Notation for depicting the use clause
CHAPTER 17

AN EXAMPLE OF THE DESIGN OF A PACKAGE SPECIFICATION IN A
HYPOTHETICAL IMPLEMENTATION

This section discusses the design of an Ada package specification using one possible implementation of this visual language. This example is not meant to establish an implementation, nor is it meant to demonstrate that there will be no implementation difficulties. It is meant to give a hint as to how such a visual language may be used and to inspire any future implementors. The implementation described here is based on a windowing system such as the X Window System.

Table 17.1 gives a small Ada package specification for stacks. This specification has been chosen because it will allow us to exercise all major options of the implementation, and yet is very simple to understand.

Figure 17.1 depicts a window which contains package specifications (it could contain other program unit specifications as well). This is a kind of library and the user may have access to more than one such library. Such a library might also depict the relationships between specifications, but this will not be discussed here.

Notice that there are various program unit icons at the bottom of the window. These will help to facilitate the design of new units. Figure 17.1 shows the user placing the pointing device over the
package specification icon.

Figure 17.2 shows that the user has "dragged" the package specification icon out to the center of the window. There the icon expands to a larger size.

Once the icon is in place and has expanded, a text window is automatically opened, as shown in Figure 17.3. This window requests the name of the new package. Once the user has finished typing the name, the return key is typed. Notice that the package name has been placed in the appropriate location in the icon.

Now the mouse button is clicked on the package specification icon. The result of this input is that a whole new window is opened; the window which corresponds to the package specification icon. It ought to be pointed out here that several such windows might be opened simultaneously, allowing the user to compare and contrast, search for items and reference the items of other packages.

In a package specification window there are four pull-down menus, scroll bars and a resize feature. There are also the depositories for language elements such as arcs. Finally, it should be noted that the package name labels the entire window.

Figure 17.4 shows the objects menu displayed. There are two types of icons in this menu, the constant icon, and the variable icon. If all that is desired is one such simple icon, then it is only necessary to "point and shoot" the desired icon. However, there are several options in choosing a constant. These options are displayed via characters. The "N" is chosen for a number constant, the "P" for a
private constant and the "L" for a limited private constant. The number constant is chosen in Figure 17.4.

Once the choice is made the appropriate icon is placed in the screen. Next, a text window requests the name and number of the object. The user first enters the name of the object, moves to the next line with a return, and finally exits the text window with a return.

Once the text window is exited the object icon displays its name as seen in Figure 17.5.

Figure 17.6 shows the type menu being accessed. Notice that all of the full type icons have associated with them three option characters: "D", "P", "L" for derived type, private type and limited type, respectively. The subtype icon shows only the derived option since the other two are not relevant for subtypes. Below the subtype icon is a listing of all of the predefined types. This allows the user to simply "point and shoot" such a type. The choice made in Figure 17.6 is for a private array.

The results of a choice of type are basically the same as was described for a choice of object. The icon is placed in the window, and a text window "pops up" and requests the name of the type.

Figure 17.7 and Figure 17.8 demonstrate the method of choosing a predefined type.

Once the FLOAT and STACK types have been placed in the window, the user may show a type dependency. By clicking on the arc icon and then clicking once on the FLOAT icon and then clicking
once on the stack icon, a type dependency arc is drawn between the two, in the order of clicks. Of course, some implementation features should defend against erroneous dependencies. The resulting dependency relationship is shown in Figure 17.9.

A pseudo-text window is displayed in the window of Figure 17.10. The window requests the description of the index constraint. One of two forms may be used. If a range identifier is to be used, the user may simply commence typing the identifier. Otherwise, the user must enter a return without having entered any other character, or click the mouse button on the discrete subtype indication icon. As before, the location for data entry is denoted by the use of black. In Figure 17.10 the discrete subtype indication method is chosen. Once chosen, the user may simply type each bound, entering them via the return key.

Figure 17.11 shows that the first bound has been entered in this manner. The second bound, however, is entered in a very different and interesting manner. This hypothetical implementation contains a feature for duplicating the icons, and placing their contents into other locations. Notice also that this is not a destructive operation; the original object MAX has remained unchanged; however, the constant MAX has been inserted as an upper bound of the index constraint.

Figure 17.12 shows that once these values have been entered, the view is returned to the high level.

The units menu is displayed in Figure 17.12. Here we see that
there are three program unit icons: subprograms, packages and tasks. The options for the former two are the "G" for generic, while the latter has the "T" option for type. However, no option is chosen in Figure 17.12 since all that is desired is an ordinary procedure.

Once chosen, the familiar sequence of placing the icon and requesting text is performed.

Again the "copy and load" facility is used in Figure 17.13. This time it is used on the STACK type; the STACK type is placed in the table for the PUSH procedure. Next, in Figure 17.14, we see that input can be placed into the table in two ways, the appropriate mode indicators (arrowheads) can be chosen for the parameter type which was just entered, or a new parameter type can be added. The copy operation is performed on the FLOAT type and it becomes the second parameter type. In Figure 17.15, we again must choose between adding additional parameters and supplying parameter modes. In Figure 17.15 parameter modes are supplied. This process of supplying parameters will continue until all parameter types are associated with at least one arrowhead and the user either enters a return or clicks the mouse button outside of the table.

The high level of the procedure PUSH has been completely defined. To access the low level the pointer is placed on the subprogram name (as shown) and the mouse button is pressed. This will result in a window similar to the one in Figure 17.16. The user need only type the formal parameter identifiers and issue returns to enter them. The first formal parameter is given the identifier S as
can be seen in 17.17. After typing the last formal parameter identifier, typing the return key brings the user back to the high level as shown in Figure 17.18.

In Figure 17.18 the mouse is pointing to the exception button. Unlike the other buttons, which are used to access pull down menus, the exception button simply results in the placement of an exception icon. This icon is shown, complete with its identifier, in Figure 17.19.

The package specification window which results from all of the previous operations is shown in Figure 17.19. In the next section, we will discuss how all of the information in this graph can be accessed.
Table 17.1 Specification for package STACK_PACK

package STACK_PACK is

    MAX : constant := 100;

    type STACK is private;

    procedure PUSH(S: in out STACK; X: in FLOAT);

    ERROR : exception;

    private
        type STACK is array (1 .. MAX) of FLOAT;
    end;

Figure 17.1 A package specification library window
Figure 17.2 Dragging the package icon into the package specification library window
Figure 17.3 The text window requesting the name of the newly created package specification
Figure 17.4 Selection of the number constant object
Figure 17.5 The number constant MAX
<table>
<thead>
<tr>
<th>Objects</th>
<th>Types</th>
<th>Units</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BOOLEAN</td>
<td>D</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>CHARACTER</td>
<td>P</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>COUNT</td>
<td>P</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>DURATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONG_FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONG_INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PRIORITY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SHORT_FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SHORT_INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSTEM.ADDRESS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SYSTEM.NAME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNIVERSAL_INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNIVERSAL_REAL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 17.6  Selection of the private array type
Figure 17.7 Selection of the predefined type FLOAT
Figure 17.8 Access of the arc operation
Figure 17.9 Use of the arc to denote the dependency of type STACK on type FLOAT
Figure 17.10 The selection of the discrete subtype indication constraint
Figure 17.11 The use of copy operation to input the upper bound of the index constraint
<table>
<thead>
<tr>
<th>Objects</th>
<th>Types</th>
<th>Units</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>

Figure 17.12 Selection of the subprogram
Figure 17.13 Use of the copy operation to define the first formal parameter to be of type STACK
Figure 17.14 Use of the copy operation to define the second formal parameter to be of type FLOAT
Figure 17.15 Determining modes of the formal parameters
Figure 17.17 Name of the first formal parameter and request to input the name of the second formal parameter
Figure 17.18 Use of Exceptions mouse button
Figure 17.19 Final visual representation of STACK_PACK package
CHAPTER 18
AN EXAMPLE OF THE REFERENCE OF A PACKAGE SPECIFICATION IN A
HYPOTHETICAL IMPLEMENTATION

This section builds upon the previous section. In the last section we described the design of an Ada package specification in a proposed implementation of the visual language. In this section we use the visual language to view the specification which was designed in the last section. Recall that, even though the language may ease implementation, its primary goal is to improve the comprehension of the package specification. In short, this chapter demonstrates the true purpose of this language.

In Figure 18.1 we see the same figure that was shown in the beginning of Chapter 17. This illustration depicts the library MY_LIB where several packages reside. Among these packages is the package which was designed in the last section; namely, STACK_PACK.

As before, we open the STACK_PACK window by clicking on the STACK_PACK package icon. The resulting window is shown in Figure 18.2 (which was taken from the end of the last section).

In Figure 18.3 the user has accessed the lower level of the MAX object by clicking on the MAX icon. The initializing icon is then displayed using the ovalar notation. The expression oval is connected to the MAX icon via the doubled arc. (This notation has been thoroughly described in previous sections.)
Next we see in Figure 18.3 that the user wishes to view the lower level of the STACK array. By clicking on the STACK icon, the index constraint is revealed. One can observe that the range of the constraint is from 1 to MAX (which, as we can see from the above access, is equal to 100). This is shown in Figure 18.4.

Next the low level of procedure push is accessed. This is done by clicking on the table cell which contains the procedure name (in this case, the name is PUSH). The result of doing so is portrayed in Figure 18.5. We see that PUSH has two parameters: one of type STACK and one of type FLOAT. We also see that the name of these formal parameters are S and X, respectively.

Since the exception has no lower level, there are no more lower levels of the packager to explore. Thus, Figure 18.5 represents the full visual equivalent of the Ada package specification of Table 17.1.
Figure 18.1 The package specification library window containing the package STACK_PACK
Figure 18.2 Accessing the low level of object MAX
Figure 18.3 Accessing the low level of type STACK
Figure 18.4 Accessing the low level of procedure PUSH
Figure 18.5 Fully expanded view of package STACK_PACK
CHAPTER 19
CONCLUSIONS

Many visual languages were reviewed during the development of this language [Chang89a, Chang89b, Kilgour89, Waite89, Wasserman90]. One of the languages reviewed was specifically for programming in Ada [SPC89]. Most of these languages did not have any provisions for describing specifications or declarations. None of them represented specifications and declarations using the detail that mine used to describe declarations. This is one of the reasons why I think that this language is unique and interesting.

During the development of this language, every attempt was made to use notations and conventions that would be meaningful. The rationales for such design decisions have been described in this thesis. Future implementors are encouraged to improve, extend and, perhaps, even to change the existing notations and conventions.

The foundation has been laid for further work. It will be the responsibility of those who continue this work to implement this language in a friendly and efficient manner. No tool is successful if it is rejected by its intended users. If a tool is perceived to be helpful in performing a task, it should not suffer this fate. Hopefully, the reader has been convinced of the merit of the ideas in this thesis. If so, then it is "ease of use" which will determine the success or failure of this visual language.
BIBLIOGRAPHY


VITA

Christopher Todd Gordon was born on October 29, 1965 in Lancaster, California.

In 1975, he and his family moved to Mays Landing, New Jersey, where he attended The Pilgrim Academy High School. He graduated from the Pilgrim Academy in May of 1983 and received his Eagle Scout award in the same year.

He attended Stockton State College in Pomona, New Jersey, and received his Bachelors of Science degree in Computer Science in 1987.

In the spring of 1987 he moved to the Washington D.C. area and worked at NASA Goddard Space Flight Center as a summer intern in 1987 and 1988.

He enrolled in the graduate program of the Computer Science Department of Virginia Polytechnic Institute and State University where he was a graduate assistant during his entire program of study. He received his Masters of Science degree in Computer Science and Applications in May of 1990.

Christopher T. Gordon

173