Data Structures for a Fragment-Based Programming Environment

Luc Pinsonneault
School of Computer Science
McGill University, Montreal

July 1987

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

Data Structures for a Fragment Based Programming Environment
Abstract

The quality of the design of program representation and supporting data structures is critically important for programming environments because the functionality of the tools in the environment is dependent on these structures. This thesis examines the issues involved in the design and implementation of the program representation and related data structures for the MUPE-2 programming environment. MUPE-2 is a fragment-based integrated programming environment for Modula-2. In particular, the typed program fragment, the structured cursor, and the cursor stack are discussed in depth. Program fragments are the basic software building blocks in the environment. The structured cursor is used, in conjunction with the cursor stack, as a basis for structured editing and error prevention in the program editor. New structures for program representation, such as that for an integrated module and partitioned data types, are also presented. An important point that emerges from this thesis is that a number of MUPE-2's tools have been simplified, or positively affected by these data structures. These tools include, amongst others, the incremental parser, the structured editor, the browser, and the semantic analyser. The structures described in this thesis are part of MUPE-2 and have been implemented in Modula-2 on a SUN-3 workstation running SUN UNIX 4.2.
Résumé

La qualité de la conception de la représentation interne des programmes et des structures de données sous-jacentes est d'une importance capitale pour les environnements de programmation. Cette importance résulte du fait que la fonctionnalité des outils de l'environnement dépend de ces structures. Cette thèse examine les issues reliées à la conception et à l'implémentation de la représentation interne des programmes, et des structures de données associées, ce pour l'environnement de programmation MUPE-2. MUPE-2 est un environnement de programmation intégré, basé sur les fragments de programmes. Les fragments de programmes "typés", le curseur structuré, et la pile de curseurs font, en particulier, l'objet d'une étude approfondie. Les fragments de programmes sont les blocs de base de construction de logiciel pour l'environnement. Le curseur structuré est utilisé, en conjonction avec la pile de curseurs, comme base pour l'édition structurée et la prévention des erreurs par l'éditeur structuré. De nouvelles structures de données incluses dans la représentation de programmes sont aussi décrites. L'objectif de ces structures est de retrouver des types de données partitionnés ainsi qu'un module intégré. Un aspect important du thème de cette thèse est que plusieurs des outils de MUPE-2 ont été simplifiés ou favorablablement influencés par ces structures de données. Parmi ces outils, citons: l'analyseur syntaxique, l'analyseur sémantique, l'éditeur structuré, et le "browser". Les structures de données décrites dans cette thèse font partie intégrante de MUPE-2. Elles ont été implémentées en Modula-2, sur une station de travail personnelle SUN-3 utilisant le système d'exploitation SUN UNIX 4.2.
Acknowledgements

This thesis represents the sum of years of studies and research. But it is only a small part of what took place in my life during these years. I hereby want to acknowledge and thank not only the persons who contributed to my academic achievements, but also those who supported and helped me surmount periods of personal crisis during these years. First, I would like to thank my parents both for making me what I am and for being who they are. I would also like to thank Dr. Nazim Madhavji for being my supervisor, and for being willing to help. Last but not least, I want to acknowledge the other members of the MUPE2 research team who all have contributed in some way to the material contained in this thesis: Kamel Toubache, Surajit Choudhury, and Rob Robson.
Table of Contents

Abstract .................................................. 1
Résumé .................................................... ii
Acknowledgements ........................................ iii
Table of Contents ........................................

Chapter 1 Introduction ...................................... 1
1.1 Related Work ........................................... 3

Chapter 2 Program Representation .......................... 5
2.1 Fragments and Fragtypes ................................ 5
2.1.1 Program Fragments .................................. 6
2.1.2 Fragtypes ........................................... 6
2.1.3 Fragment Representation .............................. 10
2.2 Programming in the Large Structures ................. 13
2.2.1 Language View vs Environment View ................ 13
2.2.2 Programming in the Large Hierarchies .............. 14
2.2.3 The Module Model vs The Definition & Implementation Module Pair 16
2.2.4 Internal Representation of other Programming in the Large Structures 19
2.2.5 The UNIT Node A Generic Construct ............... 21

2.3 Programming in the Small Structures .................. 22
2.3.1 Fine Gram Structures and Expressions ............. 23
2.3.2 Statements ......................................... 24
2.3.2.1 Simple Statements ................................ 24
2.3.2.2 Container Statements ............................ 27
2.3.2.3 If and Case Statements .......................... 28
2.3.2.4 If Elements and Case Elements ................. 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3. Declarations and Definitions</td>
<td>30</td>
</tr>
<tr>
<td>2.3.3.1. Declarations vs Definitions</td>
<td>31</td>
</tr>
<tr>
<td>2.3.3.2. Structure and Organization</td>
<td>32</td>
</tr>
<tr>
<td>2.3.3.3. Fields</td>
<td>36</td>
</tr>
<tr>
<td>2.3.4. Comments and Phrases</td>
<td>38</td>
</tr>
<tr>
<td>Chapter 3: The Structured Cursor</td>
<td>40</td>
</tr>
<tr>
<td>3.1 Cursor Types</td>
<td>40</td>
</tr>
<tr>
<td>3.1.1. The Structured Cursor</td>
<td>41</td>
</tr>
<tr>
<td>3.1.2. The Cursor Contents Classification</td>
<td>43</td>
</tr>
<tr>
<td>3.2 The Cursor Stack</td>
<td>48</td>
</tr>
<tr>
<td>3.3 Cursor Movements</td>
<td>50</td>
</tr>
<tr>
<td>3.3.1. Keyboard Originated Movements</td>
<td>51</td>
</tr>
<tr>
<td>3.3.2 Cursor Movements Using the Pointing Device: Long-Jumps</td>
<td>58</td>
</tr>
<tr>
<td>3.4. Computation of Cursor Types</td>
<td>60</td>
</tr>
<tr>
<td>Chapter 4. Conclusions</td>
<td>62</td>
</tr>
<tr>
<td>References</td>
<td>65</td>
</tr>
<tr>
<td>Appendix: MUPE-2 Architecture</td>
<td>68</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

One of the main goals of Software Engineering is to improve the software development process. This improvement is necessary because the current process cannot keep up with the ever-increasing size and complexity of software systems. One approach that offers hopes of improvement and simplification of the development process is that of Integrated Programming Environments.

Problems related to the current software development process in conventional environments include the lack of communication and of a common program representation between the different tools required to develop programs. For example, conventional environments provide editors, compilers, debuggers, and documentation systems to provide for the needs of software development. But, a programmer developing a program must enter the program using a text editor, invoke the compiler with the name of the file produced by the editor, and invoke the debugger with the name of the compiler output file. Furthermore, the programmer has to document the program as a separate activity by using the documentation system provided by the environment. It is to be noted that these tools each have a different representation for the program being developed: the editor uses text, the compiler may use parse trees, the debugger may use machine language, and the documentation system may use its own ad hoc representation.

Programming environments offer a solution to these problems by integrating the activities of all the required tools. This means that, in such an environment, tools cooperate with one another while being incrementally used. For example, while a programmer is entering a program through the environment's editor, an incremental compiler is at the same time analyzing the previous entry and generating code for it. Similarly, if an error is found while debugging a program, the cursor is placed at the location of the error, so that the programmer can immediately start editing the offending construct.
This integration is possible because all of the environment's tools share the same data structures and program representation. But, the tools' functionality is also dependent on these data structures: inadequate structures may severely limit the usefulness of some tools. For example, the program editor's cursor may move onto the different nodes in, say, the tree representation of a program. If the only tree nodes in the program representation are complete statements and declarations, then it is impossible to directly address or edit expressions or identifiers. Hence, it is important to carefully analyze the requirements of the tools in order to design the environment's data structure and program representation.

The purpose of this thesis is to discuss the issues involved in the design and implementation of the program representation and related data structures in MUPE-2. MUPE-2 is a fragment based, integrated programming environment [Mad87, Mad86c] for Modula-2 [Wir85]. It is implemented in Modula-2, on a SUN 3/50 workstation running Sun UNIX 4.2.

The structures that are considered here include: the fragment, the programming in the large and programming in the small language constructs defined by Modula-2, the structured cursor, and the cursor stack.

The author was primarily responsible for the design and implementation of the program representation and of the related structures for MUPE-2. These structures normally include all the representations for the target language constructs. In addition to these, some new structures have been designed by the author. These include:

- the fragment, which is the basic unit of program development,
- the phrase and unit nodes, which are used by the environment during the design phase of a program,
- the structured cursor, which is the program editor's unit of manipulation, and
- the cursor stack, which is used for cursor movements and for providing context sensitivity.

The author has also designed new representations for type definitions, which can be part of a type or variable declaration, and for the definition/implementation module pair. These structures are currently being implemented in the MUPE-2 system.
This thesis consists of 4 chapters. The next chapter discusses the program representation used by MUPE-2. It describes fragments, which are the basic unit of program development, programming in the large and in the small structures. Chapter 3 introduces two related data structures the structured cursor and the cursor stack. This third chapter examines the impact of the cursor and cursor stack on the environment's program editor. Finally, chapter 4 concludes the thesis.

1.1. Related Work

Program representation and supporting data structures have been one of the main topics in programming environments research.

Early syntax directed editors, such as Emily [Han71] and MENTOR [Don75], were the precursors of the modern programming environments. Being based on the syntax of the language they supported, it was natural that these editors represented programs with data structures reflecting the syntax of the language. Thus, both Emily and MENTOR represented programs with abstract syntax trees.

With the introduction of programming environments, it became more important for program representation to support all the programming activities than to mirror the syntax of the language. Thus, programming environments started evolving program representations that would support the software development process they preconize. For example, Magpie [Sch84], an integrated programming environment for Pascal, uses text to represent programs, while providing editing, parsing and static semantic analysis tools with the unit of incrementality of a single character. The parsing and semantic analysis tools generate parse trees that are used for incremental code generation, program execution and debugging tools.

It is interesting to note that Interlisp [Teit81], an environment for Lisp, does not suffer from this problem. This is because, in Lisp, the syntax of the language and that of data structures is so similar that programs can be naturally represented by these data structures.

Some environments created their own language, along with new program representation structures, in order to promote a new development process. For example, GRASP [Wor83] supports a graphical programming language represented by D-Charts. D-Charts are single entry/single exit structured control-flow diagrams that represent the...
computational structure of algorithms.

At the same time, research was also done on ways to attach semantic and contextual information to a language's grammar. This research produced two new kinds of grammars: graph grammars and attribute grammars.

Few environments, as yet, use graph grammars, as general graphs are usually more difficult to handle than trees. The IPSEN [Eng87] programming environment for Modula-2, is a notable exception. Its internal program representation is based on a graph grammar for Modula-2, and is, therefore, a general graph.

The most popular approach, in recent programming environments, is that of defining program representation from an attribute grammar of the language supported. Since attribute grammars can be represented by trees, environments using this approach represent programs by attributed syntax trees. Amongst the environments that use attributed syntax trees to represent programs are, the Cornell Program Synthesizer [Tei81b], PECAN [Rei84], Poe [Fis84], and Cépage [Mey84, Mey87].

Even some special purpose environments which support a graphical language have used attributed syntax trees to represent the program. One example of such an environment is SEGRAS [Kra86]. It was developed for writing formal specifications of concurrent and distributed application systems. It uses an algebraic and a Petri net language, and attributed syntax trees as an internal program representation.

MUPE-2 also uses attributed syntax trees for program representation. However, the grammar of Modula-2 was modified in order to introduce new constructs such as phrases and Units, and some structures in the representation have been designed specifically to support certain underlying integrated tools.
Chapter 2
Program Representation

One of the first and most important decisions during the design and implementation of an integrated programming environment such as MUPE-2 is the choice of data structures for program representation. This is because most of the facilities provided by the environment depend on these structures. For example, the data structures chosen identify all the possible cursor locations within the program. Thus, they also define the range of program segments that may be edited.

As a consequence, the requirements of all the tools of the environment have to be carefully examined before designing the program representation. For example, determining browsing breakpoints and structures that are likely to be edited in the language's grammar, is helpful in finding a comprehensive partitioning of the language's constructs. This partitioning is then used in the design of the program representation.

This chapter describes MUPE-2's data structures for program representation. It first introduces program fragments and their classification. Then structures for the programming in the large constructs of Modula-2 are discussed. Finally, the data structures for the programming in the small constructs, including statements and declarations, are described.

2.1. Fragments and Fragtypes

Existing programming environments generally use software building blocks that they can store, manipulate and modify independently, to construct programs. These building blocks, called program fragments or fragments, are syntactic units and can, depending on the programming language supported, include complete programs, procedures, modules or even statements and declarations.

The MUPE-2 approach to fragments is different from that of existing environments. This section describes the MUPE-2 approach by defining what is considered to be a fragment. Then it presents the classification of the fragments by type, and the data structure used to represent such a fragment.
2.1.1. Program Fragments

The range of fragments supported by most of these environments is restricted to the programming in the large scope because the environments strictly adhere to rules of the language they support. The language rules define the constructs that can be compiled, modified, or stored separately. These constructs are usually restricted to procedures and modules. Thus, these programming environments are bound to use the separately compilable units that the language provides as their only fragments.

For example, Magpie [Del84], and MENTOR [Don84a], which support Pascal, are restricted to work on complete programs since Pascal does not define separately compilable programming units. Similarly, Rn [Hood85a], for Fortran, and the Cornell Program Synthesizer [Tei81b], for PL/CS, can develop and store procedures, subroutines, or functions independently, since these are defined by the languages to be separately compilable.

On the other hand, MUPE-2 is a fragment based environment that integrates both the programming in the large and the programming in the small scopes of programming into a new, enlarged scope called programming in the all [Mad85c]. This integration is achieved by disregarding the language rules as to what is separately compilable, and defining, while keeping to Modula-2's syntax, a comprehensive set of program fragments extending into the new scope of programming.

Each one of these new fragments, taken in isolation, may be meaningless with regards to Modula-2 rules. For example, a conventional compiler may not even be able to recognize an isolated assignment statement before it aborts. But it is not necessary for building blocks to have a meaning of their own, since we only ask for the modules and programs assembled from these blocks to be syntactically and static semantically correct.

The range of fragments defined by MUPE-2, for the programming in the all scope, includes sequences of statements, of declarations and definitions, single expressions, procedures, modules, sequences of procedures and modules, and trees of procedures and modules.

2.1.2. Fragtypes

The fragments are classified by their contents, into categories or types, called fragtypes. This permits the environment to have a general idea of a fragment's contents without having to examine it. For example, when the environment has to check the
compatibility of two different fragments under a given operation, it can do so without having to examine the contents of both fragments.

**Meta-symbols:**

\[\begin{align*}
: &= \text{is-composed-of} \\
\langle &= \text{is-root-of} \\
| &= \text{OR} \\
\{ x \}^n &= \text{n or more occurrences of } x \\
\{ y \}^0 &= \text{null} \\
\end{align*}\]

**Fragtypes description.**

| Generic            | :=              | {Phrase | Comment} |
|--------------------|-----------------|-----------------|
| Expression         | :=              | Expression      |
| Statements         | :=              | {Phrase | Comment} Statement |
| Declarations       | :=              | {Phrase | Comment} Declaration |
| Procedure          | :=              | Procedure_template |
| Module             | :=              | Module_template |
| Unit               | :=              | Unit_template |
| DefImp Module      | :=              | Def-Imp_Module_template |
| Prog Module        | :=              | Program_Module_template |
| ProcedureSubsystem | :=              | Procedure \(\langle\{\text{Internal\_node}\}\rangle\) |
| ModuleSubsystem    | :=              | Module \(\langle\{\text{Internal\_node}\}\rangle\) |
| UnitSubsystem      | :=              | Unit \(\langle\{\text{Internal\_node}\}\rangle\) |
| DefImpSubsystem    | :=              | Def-Imp Module \(\langle\{\text{Internal\_node}\}\rangle\) |
| ProgramSubsystem   | :=              | Program Module \(\langle\{\text{Internal\_node}\}\rangle\) |
| SystemLayer        | :=              | Nodes |

<table>
<thead>
<tr>
<th>Complementary descriptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal_node</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 2.1 - Fragtypes description.

The fragtypes, described in Table 2.1, are thus used to facilitate error prevention in MUPE-2. Error prevention, in the environment, is a form of context-sensitivity which allows the environment to restrict the set of operations available to the user to those that can be applied to the current fragment. It also allows the environment to show to the user the fragments from which it is valid to choose the remaining operand. The error
Error prevention, as implemented in the environment, is made possible largely by the syntax of the commands, which requires that one operand be specified before the operation is chosen. In MUPE-2, commands have one or two operands, depending on the operation, thus the command syntax can be expressed as:

\[
\text{Command} = \text{operand} \quad \text{operation} \quad [\text{operand}]
\]

Where the first operand is always interpreted to be the current cursor. The case that is of interest now is when the cursor is on the complete fragment.

This level of context sensitivity is also made easier by the fact that it is possible, given fragtypes and operations, build tables determining what fragtypes are compatible with each fragtype/operation pair. These tables are called the fragtype compatibility tables (Table 2.2).

Although fragtypes fulfill a role similar to that of data types in a strongly type language such as Pascal or Modula-2, there exists a fundamental difference between the two. The difference is that fragtypes are dynamic attributes of fragments and are determined by the contents of the fragments, whereas data types are static attributes of data structures and determine the possible values that can be taken by the structures. For example, because no editing operation can create new fragments, the execution of such an operation involving two fragments may cause the modification of the contents and fragtype of one of the fragments involved, usually the destination fragment.

An example of such an operation is shown in Figure 2.1. A Procedure fragment is inserted inside a DefImpModule fragment, as might happen when a procedure, which has been developed independently, must be integrated within a module. The fragtype of the resulting fragment is now DefImpSubsystem.

Operations on whole fragments, such as shown above, are relatively rare since most operations will deal with only a part of the contents of a fragment. For example, inserting a statement inside a WHILE loop which is, itself, only one of the list of statements contained in a fragment of fragtype Statements.

Another important point to note is that, given the fragtypes of the operand fragments, the environment does not have to examine the fragment resulting from an
1 - Generic 6 - Module 11 - ModuleSubsystem
2 - Expression 7 - Unit 12 - UnitSubsystem
3 - Statements 8 - DefImpModule 13 - DefImpSubsystem
4 - Declarations 9 - ProgModule 14 - ProgramSubsystem
5 - Procedure 10 - ProcedureSubsystem 15 - SystemLayer

Fragtype of second operand

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: A Y in the table indicates compatibility

Table 2.2 - Fragtype compatibility table for the Insert-After operation.

operation to determine its fragtype. It is possible to compute the resulting fragtype in a manner similar to the one used for compatibility checking for each valid triple <destination-fragtype, operation, source-fragtype> found in the fragtype compatibility table, it is possible to compute the resulting fragtype. For example, inserting a Procedure fragment inside another Procedure fragment results in a fragment of fragtype Procedure Subsystem. The result of these fragtype changes is kept within a table
a) Before the operation takes place

![Diagram](image)

b) After the operation

![Diagram](image)

Figure 2.1 - Fragtype modification resulting from an Insert-Inside operation

called the *fragtype transition table* (Table 2.3).

2.1.3. Fragment Representation

In the two previous sub-sections, the concepts underlying the use of program fragments, and their classification using fragtypes, have been described. In this sub-section, the data structure used to represent program fragments in the MUPE-2 environment will be described.

As stated before, program fragments are collections of Modula-2 constructs that can be independently created and manipulated. But for the environment, we are more interested in the collection of constructs as a whole, than in the individual constructs,
Table 2.3 - Fragtype transition table for the Insert-Inside operation.

when it comes to handling the fragments. Thus it is necessary to have a container, of
glanguage constructs, that denotes a fragment in MUPE-2.

Also, we have seen that each fragment recognized by the environment has an attrib-
ute, called fragtype, which is used to label the collection of language constructs con-
tained by the fragment. The fragtype attribute is thus part of the fragment data struc-
ture.
In addition, since it is possible to store and retrieve individual fragments from a
database of fragments, called fraglib, each fragment must have some attributes that
uniquely identifies it. The environment provides some of these attributes by assigning a
unique number to every fragment that is created. This number can then be used to
retrieve the fragment. But system provided numbers are unnatural to use, so that it is
possible for the user to give a name to any particular fragment. It is not necessary that
the name be unique, since the system provided number will serve the purpose of uniquely
identifying the fragment. But a name is useful for retrieval as it permits to narrow to
number of fragments to be examined. This name is also an attribute of the fragment.

Furthermore, we allow the user to provide a description of the fragment, which is
tied to the data structure, and may be of use for some kind of retrieval based on description.

Finally, the environment may be used in a "multi-user" manner, in which case it
will be useful to keep track of such information as the name of the owner of the frag-
ment, access permissions, etc.

```
TYPE
FramePtr = POINTER TO FrameNode;
FragmentHeaderNode = RECORD
  Fragment : FramePtr; (* The frame is the actual container *)
  FragmentKind : Fragtype;
  FragmentNumber : CARDINAL,
  FragmentName,
  FragmentDescription,
  FragmentOwner : TextPtr,
  FragmentAccessPermissions : PermissionType
END;

FrameNode = RECORD
  Contents : ProgramStructuresPtr;
  CursorTrail : CursorStackPtr
END;
```

Figure 2.2 - Record structure of the fragment.
2.2. Programming in the Large Structures

The structures found at the programming in the large level include procedures, internal modules, Implementation and Definition modules, Program modules, and a non-Modula-2 construct called a Unit.

This section discusses the language view versus the environment view of such constructs; the two different amalgams composed of programming in the large constructs, namely subsystem and system-layer, the definition and implementation modules in context of the module model, and the internal representation of each individual construct.

2.2.1. Language View vs Environment View

The Modula-2 language, like most other contemporary languages, has been designed to function with conventional compilers. This means that it is organized to facilitate batch compiling. For example, only complete modules can be compiled. Also, text files are used as the medium of development of the programs.

An important restriction is created by the design of textually oriented languages. It is that the complete text of all the parts of a given compilation unit, such as declarations, procedures, and functions, must be included in the same file as the text of the "body" of the compilation unit.

This restriction has further implications for strongly typed languages such as Pascal and Modula-2. These languages require the declaration of an object to appear before the first use of the object in the program text. For example, variables and constants must be declared before they can be used. And similarly, procedures and functions must textually precede the body of the program or compilation module.

Furthermore, in Modula-2, procedures are syntactically mixed with declarations, so that anywhere a declaration can occur, so can a procedure. This can make the difference between a procedure and its declaration hard to pin down.

Because Modula-2 has been designed for a conventional textual environment, procedures and other programming in the large structures seem to be naturally confused as being only declarations. The author believes that the procedure declaration and its own "functional" part should be distinguished. This is because, even though the procedure heading can be considered a declaration, the actual implementation details of the procedure are of no concern to its users who only need to be able to correctly interface to it.
In contrast to the conventional, textual, view of Modula-2 procedures and internal modules, MUPE-2 treats programming in the large constructs almost exclusively as functional parts that appear in a hierarchy of procedures and modules. This view corresponds more closely to the top-down design of the software, as it is possible by looking at the hierarchy of procedures and modules, to see the functional decomposition of the system under examination.

However, this view of programming in the large structures is not achieved without creating some other problems. By removing the "text" of a procedure, or of an internal module, from the compilation module, the names and parameter lists of the procedures, and the name of internal modules become unavailable to the user who is browsing through the text of the compilation unit. Thus the user has to look elsewhere to find out this information.

Another related problem concerns the incremental static semantic checks for procedure calls, and uses of identifier exported from an internal module. The symbol table has to be consistent, that is contain the information about all the procedures and internal module which are part of the programming in the large structure hierarchy of the current compilation module. This consistency cannot be kept in the conventional way by only compiling the text of the module. Rather it involves a hierarchy compiler, or parser, which is going to traverse the complete procedure and module tree, and adjust all the symbol tables it encounters on the way.

2.2.2. Programming in the Large Hierarchies

The view of a program as a tree of procedures and modules is useful in that the structural information about the program functionality and design is kept with the program representation that programmers have to work with. However, this view is not adequately supported by the Modula-2 language, since the language does not provide for hierarchical or structured text. In order to represent the functional decomposition, or top-down view of a system in Modula-2, programmers have to flatten the logical tree structure of the program into a two-dimensional text file, which causes loss of visibility of the desired structure.

In MUPE-2, however, this structured view of programs and systems is provided through what are called subsystem or system-layer fragments. These fragments provide
a graphical, tree-form, view of systems, modules, and procedures which can be manipulated using an appropriate graphics tree-editor.

A subsystem fragment consists of a tree of procedures and modules representing the functional decomposition, or the top-down design of one procedure or module. This tree can be arbitrarily deep, as Modula-2 does not put a limit on the number of nesting levels for procedures and modules. The only restriction put on such a tree is that it should be syntactically legal; in Modula-2, to have all these procedures and modules nested within each other. For example, if the root of the tree is a procedure node, the user is prevented from inserting a program module node inside the tree, although he is permitted to insert the program module node as the new root of the tree. Similarly, if the root of the current tree is a separately compilable module node, the user cannot insert any other such node in the fragment.

<table>
<thead>
<tr>
<th>Modula-2 Subsystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE Fred,</td>
</tr>
<tr>
<td>VAR i: INTEGER</td>
</tr>
<tr>
<td>PROCEDURE Jim,</td>
</tr>
<tr>
<td>PROCEDURE Tom,</td>
</tr>
<tr>
<td>END Tom,</td>
</tr>
<tr>
<td>PROCEDURE Jerry,</td>
</tr>
<tr>
<td>END Jerry,</td>
</tr>
<tr>
<td>END Jim,</td>
</tr>
<tr>
<td>PROCEDURE Ted,</td>
</tr>
<tr>
<td>END Ted,</td>
</tr>
<tr>
<td>END Fred;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MUPE-2 Subsystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
</tr>
<tr>
<td>Jim</td>
</tr>
<tr>
<td>Ted</td>
</tr>
<tr>
<td>Tom</td>
</tr>
<tr>
<td>Jerry</td>
</tr>
</tbody>
</table>

Figure 23 - Subsystems in Modula-2 and in MUPE-2.

Sub-system fragments are very useful during the design, implementation and maintenance of single modules, or procedures. But, unfortunately, Modula-2 does not
provide for a hierarchical representation of programs, or systems, that require the use of several implementation modules to represent their functionality. In Modula-2, all the implementation modules, with their corresponding definition modules, are considered to be at the same hierarchical level, together with the program module.

MUPE-2 does not as yet propose to circumvent this Modula-2 shortcoming by modifying the language to allow such hierarchies of separately compilable modules. The environment, instead, provides a facility that allows a user to group together, in the same fragment, related programming in the large constructs, or subsystems. The fragment of this fragment is called system-layer because its component subsystems should form a layer of a top-down design tree.

A fragment of fragment system-layer consists of a group of subsystems. The root node of each subsystem should be syntactically compatible with the others. This means that it should be possible to find all the root nodes at the same lexical level in a Modula-2 program. For example, a system-layer containing a procedure subsystem can contain procedure, internal module and unit subsystems, but no implementation module or program module subsystem.

2.2.3. The Module Model vs The Definition & Implementation Module Pair

Theoretically, a module is a work assignment for one programmer, and it consists of an interface part, and an algorithmic or program part [Par72]. The interface part specifies those objects that the module needs to get from the outside world, the import list, and defines those that are made available to the outside world, the export list. The objects defined as exports, or imports, in the module interface can be procedures, data types, constants, or variables.

In Modula-2, this theoretical module is split in two different parts, the definition module and the implementation module, both having their own interface. The motivation behind the Modula-2 approach seems to be one of encapsulation and of protection for team development efforts. In fact, Modula-2 encourages the view of modules as black-box components of a system. For example, the user of a given module needs only be aware of what is available from the module and does not need to know the implementation details, as long as the functionality that the module provides agrees with the specifications of the module. For instance, the user of a module that implements stacks...
Figure 2.4 - System-layers in Modula-2 and in MUPE-2.

should know what procedures are provided and the number and type of the parameters, but should not care about whether stacks are implemented as arrays or as linked lists.

The definition module should be the only part of a given implementation/definition module pair to be made available to users other than the implementor. It gives to the users of the module pair the definition of the objects that are exported from the given module. It may also import from other modules the data types required for the complete definition of the objects to be exported.

The implementation module is, as its name suggests, the container of the procedures and variables the details of which are hidden from the users of the module. It is that part of the module pair that represents the program part of the theoretical module. But
it also has an interface. The implementation module interface only specifies what objects need to be imported from other module pairs, in order to be able to implement the functionality given by the specifications.

The Modula-2 facility of modules is very useful, as it provides much needed protection to the concept of module. But it also has a drawback: it is possible to unintentionally introduce inconsistencies in the module pair. In conventional environments, the definition and implementation modules are kept in two separate files. It is therefore possible, to edit one of the files and create inconsistencies that will not be discovered until compilation time.

This problem is solved, in MUPE-2, by joining the separate definition and implementation modules into one construct, called the Def-Imp module. This construct still gives to the user the dual view of the modules but eliminates inconsistencies by using the same data structure to hold the information of both modules, and displaying the information as an implementation and definition module pair [Mad86c].

This is made possible by sharing the same symbol table for both modules and by using pointers into this symbol table instead of specialized data structures, as often as possible. For example, changing the number of parameters of a procedure would, in a conventional file based environment, lead to inconsistencies between the implementation and definition modules. But, in MUPE-2, this change is applied directly to the common symbol table. Thus, because the information appears in only one place, inconsistencies of this kind can never arise in MUPE-2.

To facilitate the sharing of the symbol table, both the definition and the implementation parts of a Modula-2 module are united in the same internal node. This node has fields to point to all the required information:

- the definitions of the definition module,
- the declarations of the implementation module,
- the statements of the implementation module,
- the name and priority of the implementation module,
- the name of the definition module,
- the import lists of the definition module,
2.2.4. Internal Representation of other Programming in the Large Structures

Program Module

The program module is the only Modula-2 separately compilable module that does not come in two separate parts. This is because the program module is considered, in Modula-2, to be the driver of all the software written for a given system. Thus, while the program module needs to import procedures and data structures from other modules in order to be able to drive the execution of the system, it does not export anything to the outside world. It is a user of resources, not a producer.

Therefore, a program module node consists of fields to represent its name and priority, the import lists, the declarations, the statements, the child procedures and modules, its symbol table, and a pointer to a sibling node, if it is part of a system layer.
Internal Modules

Internal modules are similar to program modules in that they are not separated in two parts as definition/implementation modules. But they are also different in at least one aspect: they can export any object that a definition module can export, except for opaque types. These modules are mostly used to provide encapsulation of functions within a given compilation module. Thus, if the given implementation module has to provide multiple functionality, it is possible to encapsulate each of the different functions in their own internal module so as to keep with the Software Engineering goals of protection and ease of maintenance.

Since the only important syntactic difference between an internal module and a program module is that of export list, the declaration of an internal module node is the same as that of a program module, except that it has one extra field to take care of the export list.

```
InternalModuleNode = RECORD
  Header,
  ImportLists,
  ExportList,
  Declarations,
  Statements,
  Next,
  Children : AlgorithmicNodePtr;
  SymbolTable : SymbolTablePtr
END;
```

Figure 2.7 - Record structure of the internal module.
Procedures and Functions

Modula-2 procedures and functions are syntactically so similar, that they are treated structurally the same. The only syntactical difference between the two is that the header of a function has a return value type which the procedure does not have. This is easily taken care of, as the heading should be contained in the symbol table for reasons of static semantic checking. Thus, in the record describing procedures and functions, the field describing the procedure, or function, header needs only point to the symbol table entry for the header.

In this manner, the structure of a procedure and that of a function are now the same. And the record representing them contains fields for the header, the declarations, the statements, the children procedures and internal modules, the symbol table, and a pointer to a sibling.

```
ProcedureNode = RECORD
  Header,
  Declarations,
  Statements,
  Next,
  Children AlgorithmicNodePtr,
  SymbolTable SymbolTablePtr
END,
```

Figure 28 - Record structure of the procedure

2.2.5. The UNIT Node: A Generic Construct

The only programming in the large construct not described this far is the Unit node. This node has been described last for good reasons, the least of which is that it is not a Modula-2 construct.

The Unit construct is intended to be used during the design phase of a system. It is a generic node that represents a point in the design where it is not yet clear if the node should be a procedure or a module. When it becomes clear what the node should really be, then it is possible to transform the unit node into one of the Modula-2 programming in the large structures such as the ones described above.
UnitNode = RECORD
  Header,
  Declarations,
  Statements,
  Next,
  Children : ProgramStructurePtr;
  SymbolTable : SymbolTablePtr
END;

Figure 29 - Record structure of the Unit node.

Given that the data structure representing a Unit node is very simple, the transformation is not really a difficult one. To explain this further, a unit node is basically the intersection of a module node and of a procedure node. Thus, it contains fields for the unit name, declarations, statements, siblings, and children. It does not, however, contain any information about interface as the specification of module and procedure interfaces produce an empty intersection. Hence, a transformation operation only involves creating the appropriate node and to reassign the fields of the Unit node.

One last point about the Unit construct: MUPE-2 is a programming environment now, but plans are being made to extend it into a software engineering environment. Thus it would become necessary to keep the design history of the systems developed in MUPE-2. Since the Unit node is used at design time, clearly it is part of the design history of a system. Therefore, it would have to be kept in some form in the history of a system after it has been transformed.

2.3. Programming in the Small Structures

The structures found at the programming in the small level include: expressions, statements, declarations, definitions, comments, and a non-Modula-2 construct called a phrase.

This section discusses the representation of expressions, selection statements, other Modula-2 statements, the differences between definitions and declarations and their impact on the declarations fragtype, and, finally, the representation and meaning of the phrase construct.
The transformation of the Unit:

UNIT example;
VAR
  i : INTEGER
BEGIN
  ReadInt( i );

END example,

into the module:

MODULE example,
VAR
  i : INTEGER
BEGIN
  ReadInt( i ),

END example,

is easily achieved by following this simple algorithm:

NEW(p,InternalModuleNode), (* create an internal module node *)
WITH p↑ DO,
  Header = Unit↑ Header,
  Declarations = Unit↑ Declarations;
  Statements := Unit↑ Statements;
  Next := Unit↑ Next,
  Children := Unit↑ Children,
  SymbolTable := Unit↑ SymbolTable;
  ImportLists := NIL,
  ExportList := NIL,
END,

Figure 2.10 - Transformation of a Unit into a Module

2.3.1. Fine Grain Structures and Expressions

In MUPE-2, variables, expressions, lists of expressions, lists consisting of expressions and subranges, and designators are all grouped under one category of structures called fine grain structures. These structures are called fine grain because they are at a fine enough level of granularity that it was not felt necessary to allow structured cursor movement within them.

Fine grain structures are the smallest granularity of structures that can directly form a fragment. The fragment type of a fragment containing only a fine grain structure is
Expression.

These structures can only be edited in a textual manner. This is because it is simpler and more natural to enter and edit expressions in a textual manner, than to do the same in a structured manner on the abstract syntax tree (AST) or the parse tree of the expression. This textual view of expressions and other fine grain structures is not unique to MUPE-2 Other environments such as the Cornell Program Synthesizer [Tei81b] and PECAN [Rei84b] also treat some program constructs, such as expressions and assignment statement, in a textual manner for the reasons stated above. One disadvantage of treating fine grain structures in textual mode is that every time the text is modified, the environment has to reparse the complete fine grain structure instead of only parsing that part which has been modified. This parsing will be taken care of by an expression parser which is currently under development.

Expressions and other fine grain structures are internally represented by an AST. No identifier is present in the AST. Instead, for efficiency in storage, unparsing, and global changes, they are kept within the symbol table. Constants, unless they belong to an enumeration type, are kept within the AST.

2.3.2. Statements

In MUPE-2, the statements are divided into three categories: simple one line statements, containers of a statement sequence, and containers of a list of statement sequences. Examples of the three categories include the Assignment statement, the While loop statement, and the Case statement, respectively.

This division is necessary because, although statements of all three categories can appear in the same fragment, they are treated differently by the editor depending on the category they belong to. This behavior of the editor can be noticed, for instance, in the different options offered to the user for the Insert command, when the cursor is moved on statements of the different categories.

2.3.2.1. Simple Statements

The simple, one line, statements provided by Modula-2 are the Assignment, ProcedureCall, Return, and Exit statements. These statements are called simple because they do not contain other statements, and will, in general, be treated as fine grain structures.
Using the previously declared structures to define the AST nodes will translate the expression \( a \uparrow b * a \uparrow c \) into the following AST. It is assumed that the identifiers represent links to the symbol table.

Figure 2.11 - Fine grain structures and declarations.
These simple statements do not generally lend themselves to entry by template selection, mainly because it is quicker and more convenient to enter them textually, directly from the keyboard. For these same reasons, the Synthesizer forces textual entry and editing of the Assignment, ProcedureCall and Return statements. However, in MUPE-2, the Return, and Exit statements can be entered by template selection because they both consist only of one reserved word in most of the cases. Also, in some cases it might be faster to choose the template from a menu than to type the statement in at the keyboard.

![Figure 2.12 - Record structure of simple statements.](image)

Other consequences of simple statements being treated as fine grain structures are that the structured cursor cannot be moved within such a statement and that modifying such a statement can only be done in a textual manner. An exception to these rules is the assignment statement. Since this statement is composed of two parts, a designator and
an expression, it can be treated in a structured manner by allowing the cursor to move onto either side of the ":=" operator. Thus modifications can be made to only one part of the statement instead of the whole statement.

2.3.2.2. Container Statements

The container statements provided by Modula-2 are the While, For, Repeat, Loop, and With statements. They are called "containers" because they can enclose a sequence of statements. For the most part, these statements represent the control structures of the language.

```
WhileStatement = RECORD
    WhileExpression : ExpressionPtr,
    Contents,
    NextStatement : ProgramStructurePtr
END

RepeatStmt = RECORD
    RepeatExpr : ExpressionPtr;
    Contents,
    NextStatement : ProgramStructurePtr
END

ForStmt = RECORD
    ForExprs : ExpressionPtr;
    Contents,
    NextStatement : ProgramStructurePtr
END

LoopStmt = RECORD
    Contents,
    NextStatement : ProgramStructurePtr
END

WithStmt = RECORD
    WithDesignator : ExpressionPtr;
    Contents,
    NextStatement : ProgramStructurePtr
END
```

Figure 2.13 - Record structure of the container statements.

These statements will usually be created using templates selected from a menu, as it is more convenient to have the system provide the bracketing of such constructs. It will also be possible to textually enter new container statements and modify existing ones by
using the system's text editor and the fragment parser.

Since container statements lexically include a sequence of statements, and that it is possible to position the structured cursor on any statement existing within the environment, structured movements of the cursor are allowed inside the container statements. Furthermore, since most of these statements include an expression, or designator for the With statement, it is also possible to move the structured cursor on the complete expression.

2.3.2.3. If and Case Statements

The If and Case statements are not treated like containers statements because, unlike containers, they do not directly contain a sequence of statements. Rather, they encompass a sequence of If or Case elements, respectively. An If (Case) element is that part of the If (Case) statement which contains the selection condition and the statements to be executed if the condition is met. The MUPE-2 view of the If and Case statements is different from the conventional one. Usually, a selection statement such as If, is viewed either directly as a sequence of elements without a container, or as a nested structure of binary selections. For example the nested IF-THEN-ELSE statements of Pascal achieving an effect similar to the sequential IF-THEN-ELSIF-ELSE of Modula-2. In MUPE-2, we choose to see these statements as containers of If and Case elements where the first element (of an If) corresponds to the THEN-part and the last one may correspond to the ELSE-part.

One of the reason for interpreting If and Case statements as containers of If and Case elements, respectively, is browsing. It was found attractive to be able to move the cursor directly from one If (Case) element to another element, within a given If (Case) statement. Also, this interpretation allows for more interesting manipulations to be performed on the elements of these statements; these manipulations are described in the following sub-section. This is clearly not possible with conventional views of the If and Case statements.

For reasons similar to those of container statements, new If or Case statements are usually created by template selection, and structured cursor movements are allowed inside the If and the Case statements. Furthermore, the Case statement contains an expression which is not part of any of the clauses it contains, so that it is possible to
move the cursor onto the Case expression.

```plaintext
IFStatement = RECORD
  Contents,
  FirstIfElement,
  LastIfElement,
  NextStatement  ProgramStructurePtr
END,
(* FirstIfElement and LastIfElement are used by the environment
to interpret the different ifElements *)

CaseStatement = RECORD
  CaseExpr  ExpressionPtr,
  Contents,
  LastCaseElement,
  NextStatement  ProgramStructurePtr
END,
(* LastCaseElement is used by the environment to interpret
the different CaseElements *)
```

Figure 2.14 - Record structure of selection statements

2.3.2.4. If Elements and Case Elements

As stated before, If (Case) elements are those parts of an If (Case) statement that include a conditional test and the statement sequence to be executed if the condition is true. This makes the If and Case elements in many points similar to container statements. But there is still one important difference: the If and Case elements are not statements. Therefore they cannot exist outside of an actual If or Case statement.

Structurally, all the If (Case) elements within an If (Case) statement are exactly the same. There is no such structure as a THEN If element or an ELSE If element. The environment interprets the meaning of the various elements depending on their position in the sequence of elements and on whether they actually contain a conditional expression or not. For example, if the last element of an If statement contains an expression, it will be interpreted by the environment as an ELSIF part of the If. However, if the element did not contain any expression, it would be interpreted as an ELSE. The interpretation is similar in the case of a Case statement. There is one special case that is handled differently, depending on whether the container is a Case or an If statement, and this case occurs when there is only one element within the statement. This is when the
element is both the first and the last one in the sequence and does not contain an expression. In this case, the element is interpreted as an ELSE part if the containing statement is a Case. However, if the containing statement is an If, then the element has to be interpreted as a THEN part, since it is syntactically incorrect to be missing the THEN part in an If statement.

The fact that the interpretation of If and Case elements is mainly dependent on the position of the elements within the sequence has introduced a rather interesting side-effect. It is now possible to freely rearrange the order of the elements of a particular If statement, for example, without introducing any syntactic problems. It is also possible to move or copy elements from one If statement to another, regardless of the initial interpretation. For example, the user could move the ELSE part of an If statement to the position before the THEN part of another If without causing any clash. This is because, as soon as the move is completed, the old ELSE will be interpreted as the new THEN part and the old THEN will be reinterpreted as either an ELSE or an ELSIF part, depending on whether it actually contains an expression or not.

```
CaseElement = RECORD
  CaseLabelList ExpressionPtr;
  Contents,
  NextCaseElement : ProgramStructurePtr
  END,

IfElement = RECORD
  IfExpr : ExpressionPtr;
  Contents,
  NextIfElement : ProgramStructurePtr
  END,
```

Figure 2.15 - Record structure of the If and Case elements.

2.3.3. Declarations and Definitions

This section discusses the structures used in MUPE-2 to represent Modula-2 declarations and definition. And it shows that, even though the Modula-2 syntax makes them different objects, it is possible to treat them uniformly, as long as they are not placed in the context of one of the programming in the large constructs. This consideration simplifies the design of operations on declarations and definitions.
2.3.3.1. Declarations vs Definitions

When examined out of context, declarations and definitions are nearly identical. For example, variables and constants declarations are syntactically the same as variables and constants definitions, respectively. Furthermore, the only difference between type declarations and type definitions is that type definitions may include opaque types.

<table>
<thead>
<tr>
<th>Declarations</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CONST} &lt;id&gt; := &lt;ConstExpr&gt;</td>
<td>&lt;id&gt; := &lt;ConstExpr&gt;</td>
</tr>
<tr>
<td>\texttt{VAR} &lt;idlist&gt; := &lt;type&gt;</td>
<td>&lt;idlist&gt; := &lt;type&gt;</td>
</tr>
<tr>
<td>\texttt{TYPE} &lt;id&gt; := &lt;type&gt;</td>
<td>&lt;id&gt; [ := &lt;type&gt; ]</td>
</tr>
<tr>
<td>\texttt{PROCEDURE} &lt;id&gt; := &lt;type&gt;</td>
<td>\texttt{PROCEDURE} &lt;id&gt; [&lt;FormalParameterList&gt;]</td>
</tr>
</tbody>
</table>

Figure 2.16 - Similarities between the syntax of declarations and definitions

The ultimate difference between declarations and definitions is that definitions include procedure headings, whereas, since MUPE-2 represents procedures in a structured manner, declarations do not even include procedures. It is thus apparent that definitions, in MUPE-2, are a super-set of the declarations, in that any declaration can be a definition, but not all definitions can be treated as declarations.

Thus, it is impossible, in a context-free situation, to distinguish declarations from definitions if no opaque types or procedure headings are used. This is why, in the environment, no distinctions are made between definitions and declarations when they are in a context-free situation such as a Declarations fragment. But as soon as they are placed in a context, for example, the Declarations fragment is inserted within a procedure fragment, it becomes important, both syntactically and semantically, to differentiate between the two.

Unfortunately, because the environment does not syntactically make the difference between definitions and declarations when they are in a context-free situation, making the distinction, when within a context, involves semantics checks. For example, if a Declarations fragment is moved inside a procedure, the environment cannot accept the moved fragment until it checks it for the absence of opaque types and procedure headings. To facilitate these checks, the declaration blocks in a fragment have counters to indicate the number of opaque types and other kind of declarations. Also, counts are
procedure headings. Since the fragment has already been parsed and accepted as syntactically correct, the environment now has to look at the counters of the main structures of the fragment and decide whether it is acceptable in the context of a procedure. If there are procedure headings in the Declarations fragment, then checking only the procedure heading counter of the frame is sufficient to determine the acceptability of the fragment. Otherwise, the declaration blocks should be examined, possibly exhaustively, to prove the absence of opaque types in the fragment. There obviously is no such problem in the context of a definition module since it can accept a full definition list.

2.3.3.2. Structure and Organization

While it would be possible to use only the symbol table to represent the declarations and definitions within the environment, it is not practical to do so. This is because of the loss of positional information, such as the order of the user typed declarations, and of difficulties in unparsing, and in structured editing of the declarations. Thus, although it is not necessary from the compilation point of view, the environment uses data structures, in addition to the symbol table, to represent the different declarations and definitions.

The structure of the definitions and declarations used by the environment reflects the syntax of Modula-2. That is, a definition list is a list of type, variable, and constant blocks intermixed with procedure headings; whereas a declaration list excludes procedure headings and opaque type definitions.

Declaration, or definition, blocks such as the ones given above, correspond to the syntax of declarations. That is, a type block corresponds to the reserved word TYPE in the grammar, and the block contains the type declarations that would follow the word TYPE in the language. The main difference between the environment's representation of declaration blocks and the language's syntax is that, following the syntax, the reserved word only means that the following declarations, up to the next reserved word, are, say, type declarations; whereas, MUPE-2 treats blocks as containers of declarations. This treatment facilitates structured editing operations and unparsing for the environment, although it removes some flexibility from the user, as it makes it practically impossible to freely move declarations around.
TypeDeclBlock = RECORD
  NumOfOpaqueTypesInBlock : CARDINAL;
  TypeDeclarations,
  NextDeclBlock : ProgramStructurePtr
  END;

VarDeclBlock = RECORD
  VarDeclarations,
  NextDeclBlock : ProgramStructurePtr
  END;

ConstDeclBlock = RECORD
  ConstDeclarations,
  NextDeclBlock : ProgramStructurePtr
  END;

Figure 2.17 - Record structure of the various declaration blocks.

Procedure headings, while they are at the same level as declaration blocks in the
definition list, are not nearly as complicated to handle. They consist of a pointer to the
symbol table entry for the procedure they represent. This makes it impossible to have
name or type conflicts between a procedure heading in a definition module and the actual
procedure in the implementation module. In fact, any change to the procedure interface,
whether it is made in the definition module or in the implementation module, will be
immediately reflected at both places, thus avoiding any temporary inconsistencies. For
this reason, it is to be noted that procedure headings can only be edited in textual mode,
since there is no AST representation for them.

ProcedureHeading = RECORD
  NameAndParas : SymbolTablePtr,
  NextDeclBlock : ProgramStructurePtr
  END;

Figure 2.18 - Record structure of the procedure heading definition.

As for declarations themselves, including opaque types, they also mostly correspond
to the syntax of Modula-2. All the declarations, except the opaque type which is atomic,
are structured constructs reflecting their syntax. For example, a constant declaration is
represented by a binary tree node, of which one child is the constant identifier, and the
other is the constant value or expression. The declarations have been structured in such a
way as to simplify the editing of parts of a declaration at a time, and to have structured
cursor movement within each declaration.

ConstantDeclaration = RECORD
  ConstId,
  ConstExpr : ExpressionPtr;
  NextDeclaration : ProgramStructurePtr
END;

VariableDeclaration = RECORD
  Vars: ExpressionPtr,
  VarTypeDef : TypePartitionPtr,
  NextDeclaration : ProgramStructurePtr
END;

TypeDeclaration = RECORD
  TypeId ExpressionPtr;
  TypeDef TypePartitionPtr,
  NextDeclaration : ProgramStructurePtr
END;

OpaqueType = RECORD
  OpaqueTypeId FineGrainPtr;
  NextDeclaration : ProgramStructurePtr
END,

Figure 2.19 - Record structure of the declarations

As mentioned above, a constant declaration is a two parts structure consisting of an
identifier and an expression. The variable and type declarations are similar to the con-
stant declaration in that they are also two parts structures. The first part is an identifier
list for variable declarations and a single identifier for type declarations. However, both
the variable and type declarations differ from the constant declarations in that their
second part is also structured, as it is a list of type partitions. This means that not only
is the user capable of moving the cursor on the left hand side of a variable or type
declaration, but he can move it on different elements of the partition list of the right
hand side part.

This is because the environment must be able to represent, for example, an array of
pointers to records in a manner that will allow the user to:

- modify independently the array subscripts and the record fields of the declaration;
or

or
change the declaration to a pointer to an array of records in only one operation.

This partitioning of type definitions is unique to MUPE-2 as other environments allow either only structured movements or textual movements within a type definition. For example, Gandolf and Cépage treat type definitions in a structured manner, allowing only structured movement, while Magpie treats them as only more text. This implies that the modification of type definitions in such environments is not trivial as it may involve multiple operations from the part of the user as well as extensive reparsing and semantic analysis. In contrast, MUPE-2's type partitions decrease both the number of operations required from the user and the complexity of the reparsing and semantic analysis following a change in the type definition.

There are as many different partitions as there are types in the Modula-2 syntax. This means that the type partitions are SimpleType, ArrayType, RecordType, SetType, PointerType, and ProcedureType.

All type partition lists end with either a SimpleType or a RecordType partition. The SimpleType partition holds the place of one of the three simple types identified in the grammar of the language: unqualified, subrange, or enumeration. It is represented, in the environment, by a pointer into the symbol table if the current type is either a qualdent or an enumeration type. Otherwise, it is a pointer to a fine-grain construct to represent the subrange.

The RecordType partition is the most complex of all the partitions, as it is itself a highly structured construct, so that it is possible to achieve structured cursor movements within it. It is represented as a container for a list of fields and case variants.

The SetType and PointerType partitions are the simplest yet, since all they stand for are the reserved words SET OF and POINTER TO, respectively. Thus, internally they are only pointers to the next partition.

The ArrayType partition is also quite simple, although it may not seem so. It is simply a pointer to a list of simple types which form its indices.

Similarly, the ProcedureType partition is also very simply represented. By not allowing any structured operation within this partition, it is possible to represent it simply as a pointer into the symbol table.
PartitionType = (SimpleTypes, Sets, Arrays, Pointers, Records, Procs)
TypePartition = RECORD
  CASE PartitionKind : PartitionType OF
    SimpleTypes :
      SimpleType : FineGramPtr |
    Sets :
      SetType : ProgramStructurePtr |
    Arrays :
      Indices : FineGramPtr;
      BaseType : ProgramStructurePtr |
    Pointers
      PointerType : ProgramStructurePtr |
    Records
      Fields : ProgramStructurePtr |
    Procs
      ProcsPara : FineGramPtr,
      FuncType : SymbolTablePtr
    END |
  NextPartition : ProgramStructurePtr
END |

Figure 2.20 - Record structure of the type partitions.

2.3.3.3. Fields

The fields of a record can be classified under two categories: the fixed fields, which look like variable declarations, and the variant fields, which look more like case statements.

Field = RECORD
  FieldIds,
  FieldTypeDef,
  NextField : ProgramStructurePtr
END |

Figure 2.21 - Record structure of fixed fields.

The fixed fields are very simple to represent as their syntax is exactly the same as variable declarations. Thus, fixed fields are structured constructs that have on the left
hand side an identifier list and on the right hand side, a list of type partitions.

Variant fields, on the other hand, are very much like case statements. They are containers of variant elements and they have a selector, which is different from an expression, on which the cursor can be positioned.

The selector, in its general form, is similar to a variable declaration, except that the left hand side identifier is optional and that the list of type partitions is restricted to a qualifier. The differences are made necessary by the grammar of Modula-2, which defines the selector as having an optional field identifier and a qualifier for type. Even with this similarity, the environment represents the selector not as a structured construct, but as a pointer into the symbol table. This is because it was found more convenient to enter or edit a selector in a textual manner.

The variant elements resemble the case elements used for the Case statement. They have a label list which is syntactically identical, and they contain a field sequence. Their structure is an image of the structure of the Case elements. And this resemblance is not an accident. It allows the cursor to move directly from one variant element to the next, and to allow the free reordering of the different elements. As with case elements, the interpretation of variant elements is dependent on the position within the sequence and on the existence of an actual label list.

```
VariantField = RECORD
  CaseSelector,
  LastCaseVariantElement,
  ContainedCaseVariantElements,
  NextField ProgramStructurePtr
END.

CaseVariantElement = RECORD
  CaseVariantLabelList,
  ContainedFields,
  NextCaseVariantElement ProgramStructurePtr
END.
```

*Figure 2.22 - Record structure of variant fields and variant elements*
2.3.4. Comments and Phrases

The other two structures used while programming in the small are the comment and the phrase. These two structures have in common that they both can appear in the middle of declarations and statements, but their purposes are different.

The comments are there to serve their usual role of in-text program documentation. The environment treats them as actual language constructs, except that they are not executable and will be omitted during code generation. The syntax of Modula-2 comments had to be modified to allow them to be parsed and kept at various lexical levels in the internal representation of a program fragment.

The actual implementation of comments is a pointer to a memory block of text, not restricted in size. Future plans are to extend this implementation to include graphics (à la MacPaint).

Comment = RECORD
  Body : TextPtr;
  Next : ProgramStructurePtr
END;

Figure 2.23 - Record structure of comments

Phrases are for programming in the small what Units are for programming in the large. They serve as design decision points, so that they could be treated as generic statements, or declarations, or expressions. They are kept in the development history of the current fragment, so as to help keep the design/implementation history documentation of a system.

Phrase = RECORD
  Body : TextPtr;
  Next : ProgramStructurePtr
END;

Figure 2.24 - Record structure of phrases

In contrast to comments, phrases are not disregarded during the execution of a fragment. When the interpreter reaches a phrase, the execution is interrupted and control is
returned to the editor, leaving the cursor on the phrase which interrupted the execution. Again, the syntax of Modula-2 had to be modified to allow for phrases. The current implementation of phrases is the same as a comment. It is a pointer to a memory block of text which is not restricted in size.
Chapter 3
The Structured Cursor

A cursor is a mechanism that is used to focus the attention of a user on a given portion of a computer display. In conventional text editors [Joy80, Sta81] and in some programming environments [DeI83, Tel81b], the cursor focuses the user's attention on a character position. In contrast, the MUPE-2 cursor focuses the attention of the user on complete program structures.

Whereas cursors are usually used only for display purposes, in MUPE-2 the cursor is also the basis of all editing and browsing operations. This is because, in contrast to usual cursors, the environment's cursor is not only the representation of a position within program text, but also is the representation of the actual program structure on which it is positioned. This is why we call the cursor structured.

This chapter will discuss, in order, the following points: the classification of cursors by their contents and the cursor representation, the cursor stack that represents the context of the cursor, the manner in which cursor movements are achieved in MUPE-2, and the method used for computing the cursor types.

3.1. Cursor Types

Because the cursor represents the program structure on which it is positioned, it is used by the environment to determine the compatibility of the operands of editing operations. But, the cursor by itself does not provide enough information to perform these checks. This is why the cursor need to be attributed with information about its contents. This information is then used to classify the cursor into a precise category.
This section will define what the cursor is and how it is represented, first, and then describe the cursor classification scheme.

3.1.1. The Structured Cursor

A cursor is a marker that attracts and focuses the attention of the user on a precise part of a device. In the case that is of immediate interest, that is for a programming environment, the device is the screen of a computer terminal or of a monitor. The cursor, in this case, focuses the attention of the user on that part of the screen where the next action is going to take place. In the case of graphics-oriented applications, the term cursor is replaced by pointer, as the current technology permits the use of pointing devices such as light pen, mouse, joystick, graphics tablet, touch screen, etc.

In addition to focusing the attention of a user to one part of the screen, the cursor often has an implicit meaning. For example, when working with a full-screen text editor on an alpha-numeric terminal, the cursor may represent different things. If the user is inserting text, then the cursor usually represents the character before which all the input text will be placed. Whereas, if the user is deleting, or modifying text, it represents the character that is to be deleted, or modified, next.

Programming environments, in general, provide program editors instead of text editors. Or more specifically, they provide structured program editors, which take advantage of the structure inherent in programming languages to simplify program entry for the users. Since these editors work with structured objects, namely programs, it is obvious that even though they must use a cursor to focus the attention of the user, the meanings associated with the cursor must be different than those used with unstructured text. For example, the cursor in a program environment may represent a complete procedure, which is itself displayed on the terminal screen as many lines of text. In this case, it is obviously meaningless to equate the cursor with the first character in the first line of the procedure.

However, the meaning given to the cursor, in a program editor, depends largely on the manner the implementors view programs. The program views covered by existing programming environments range from purely textual, as in Magpie [Del84], to completely structural, or syntax-driven, as in the Gandalf project [Nol85]. So that in
Magpie, where the editor provided is a full screen text editor with extra functionality, the cursor can be said to only represent the character it is positioned on. Whereas in Gandalf, the cursor always represent a structure, whether an identifier or a procedure. Notice that even though Gandalf tries to view programs as pure structures, they still have to allow some amount of textual interaction to permit users to type in identifiers and variable names.

However, the majority of the environments provide a program view that is somewhere between these two extremes. For example, the Cornell Program Synthesizer [Tei81b] provides structure down to the expression level, but certain items like the Assignment and ProcedureCall statements, and expressions are entered textually. Thus, it is possible for the cursor to represent either single characters or complete structures within the same editor. It also becomes important to know exactly what the cursor represents, both for the system and for the user, when using such an editor.

A different approach was taken in both IPSEN [Eng87], PECAN [Rei84a], and MUPE-2, instead of trying to find a comfortable middle point between the two extreme views of programs, these environments try to accommodate both. This means that it is possible, in these environments to view a program as an, almost, pure structure or as pure text, or anything in between. The main difference between MUPE-2 and IPSEN, or PECAN, is that MUPE-2 allows textual editing of a sequence of syntactic entities while both IPSEN and PECAN are restricted to one such entity.

To achieve this, MUPE-2 provides what could be considered two separate editors, but is really a two mode editor. The two modes are, as opposed to conventional editor modes (command & input), textual and structural modes. In most cases, the switch from mode to mode is automatic, textual mode being triggered by keyboard input and structural mode being triggered by the use of the pointing device or an explicit command to exit the textual mode.

Because of these two modes, which coincide with the two possible interpretations of the cursor, and because of the presence of a pointing device, MUPE-2 makes the distinction between three different kinds of cursors. They are:

- the textual cursor, which refers to a character position within a text,
- the structured cursor, which refers to a structure in the structural view of programs, and
- the pointer, activated by the pointing device and representing a pixel position on the screen.
Because both the textual cursor and the pointer represent only the position of unstructured objects, it is not in the scope of this thesis to discuss them. But the structured cursor definitely is.

In MUPE-2, the structured cursor is considered to be not only a highlighted region of the display screen, but also the equivalent of the actual structure on which it is positioned. This allows the environment to provide operations on the actual cursor, where the cursor is used both as an operand by the operation and to display the chosen structure to the user.

The cursor is implemented as a record consisting of two fields: a pointer to the structure on which the cursor is positioned and its cursortype. Cursortypes are described in the next section.

```pascal
CursorNode = RECORD
  Cursor : ProgramStructurePtr,
  Cursortype : CursorKinds
END;
```

Figure 3.1 - Record structure of the cursor

3.1.2. The Cursor Contents Classification

There is an interesting comparison to be made between the structured cursor and program fragments. One part of it is that fragments are containers of program structures and are used as operands of operations for the complete development phase. That is that it is possible to run fragments, debug them, and then assemble them, using the editor, to form a complete program. The other part is that the cursor is used as an operand to perform all the operations within fragments, but cannot be run, although it is used for editing purposes while debugging a particular fragment. Another interesting point is that each fragment contained in the environment contains at least one cursor.

It is estimated that, in the development phase of a program, operations involving complete fragments will be less frequent than those taking place inside those same fragments, because the majority of them should be editing operations. Also, it is possible in MUPE-2 to extend the cursor to contain, or represent, the complete contents of a whole fragment. In this case, ideally, the cursor is, and represents, the fragment. Note that this
does not mean that when the cursor is deleted, the fragment loses its identity, but rather that it is possible to use the complete contents of a fragment in an editing operation without having to lose the fragment's identity.

Similarly to fragments, and fragtypes, which are the basis for the environment [Mad87], it can be said that the structured cursor is the basis for the MUPE-2 structured editor. This is because most editing operations use the cursor as an operand. The only exception is when a complete fragment is selected, via mouse selection, as one of the operands. Note that an editor operation, such as insert, or move, involving a fragment selected with the mouse or the cursor, really involves only the contents of the fragment. it does not affect the identity of the fragment, although it may involve a fragtype change.

Another similarity between the structured cursor and fragments is that the cursor receives a type every time it moves to a new structure. This type is mainly determined by the structures contained by the cursor. The types given to the structured cursor are called cursortypes. In contrast to fragtypes, cursortypes do not only depend on the nature of the contained structures, but also, to a high degree, on the context of these structures. An example of context dependent cursortypes is given in Figure 3.2.

```
VAR
Counter, Limit INTEGER,
ch CHAR.
{"Limit is a soft bound on the number
of chars to be processed in one pass *)

REPEAT
ReadChar(ch),
Counter = Counter + 1,
{"Read n = Limit chars and send them
to the analyser *)
UNTIL Counter >= Limit;
```

Figure 3.2 - Examples of context dependent cursortypes.

Consider the two comments outlined above. It is important, when using one of the comments as an operand for an editing operation, to know that a given comment is within a list of declarations or of statements. This is because, although comments can appear in both declarations and statements, they cannot arbitrarily be replaced by structures they are compatible with. For example, the comment in the declaration list could be moved in the statement list, but should never be replaced by a statement, in its current context, as statements cannot appear in the middle of declarations.
There are cases where the cursortype may seem to depend only on the contents of the cursor. For instance, if the cursor is containing an assignment statement, it seems unnecessary to consult the context to determine the cursortype, as the node contained by the cursor indicates that it is an assignment statement. Furthermore, because the environment ensures syntactic correctness, the cursor can only be in a fragment of type Statements or in the statement list of a procedure or a module.

However, as soon as operations between this statement and programming in the large structures are being considered, context becomes very important. This occurs because such operations are valid only under special circumstances. For example, if the statement is part of a list of statements in a fragment, replacing it by a procedure node, or inserting such a node around the statement is syntactically incorrect. The same is true if the statement is the only statement of a procedure or module. The only circumstance where these operations are legal is when the statement is the only structure contained by a fragment. Thus, the cursortype should indicate whether the statement is part of a list, or is in a procedure or module, or possibly the only structure in a fragment, in order to permit sensible operations between programming in the large structures and statements.

In addition to compatibility checks for editing operations, the cursortype context is used for several other purposes in the environment, such as error prevention, parsing of expressions and textually modified structures, and unparsing structures which are to be re-interpreted after an editing operation. Examples illustrating and clarifying these uses follow.

- Based on the cursor and its context, the environment can determine which are the commands and/or options that are available to the user. In this way, the possibility of an erroneous choice of command and/or option is removed. But error prevention does not stop at this point. Given a partially complete operation specification, one for which one operand, the command and option are given, the environment will show the user, by coloring the cursor, if the cursor contents could legally be used as an operand to complete the operation specification.

- In some cases, the information provided by the contents of the cursor is not sufficient to determine if the operands of an editing operation are syntactically compatible. Hence, in such cases, it is not guaranteed that the result of the operation...
will be syntactically correct. For example, assume that the cursor is on the *simple type* partition of a variable declaration. If the partition is the only one in the declaration, then it can be replaced by any partition, or list of partitions. But, if the cursor was on a simple type partition immediately preceded by a SET OF partition, the result of an arbitrary replacement could very well be syntactically incorrect since the Modula-2 grammar states that only a simple type can follow the SET OF reserved words. Thus, the fact that the simple type partition is preceded by the SET OF partition, its context must be recorded in the cursortype for the compatibility checks to ensure the syntactic correctness of the operation's result.

The environment provides incremental compilation during the editing stages of program development. And because any structure on which the cursor can be positioned can be textually edited, it is necessary to include the context of the cursor to the parser in order to trigger the appropriate production and semantic checks when parsing the modified contents of the cursor. For example, if the cursor is positioned on the expression of a While or a Repeat loop, the expression should evaluate to a boolean value. However, if the only information available to the incremental compiler is that the cursor contains an expression, then any expression would be syntactically valid as a result. This means that the static semantic checks could not be performed completely by the parser, since the final type of the expression does not matter when evaluating a context-free expression. Similarly, if the cursor was positioned on the selection expression of a Case statement, and if the cursortype only reported that the cursor was on an expression, then an expression that evaluates to a value of type REAL would be considered semantically legal to replace it. Thus, providing contextual information as part of the fragment simplifies the process of incremental compilation within the environment.

Finally, there are structures within the environment, whose interpretation depends on their position in a sequence. These structures, as introduced in the previous chapter, are the Case and If elements. Because of this positional dependency, it is obvious that the cursortype of a cursor positioned on such a structure cannot be obtained by only examining the structures it contains. For example, if the cursor is positioned on an if element, the environment cannot decide if the cursortype should be THEN-part, ELSIF-part, or ELSE-part without examining the complete sequence of if elements, i.e. the context. In this case, the context is needed for
unparsing reasons, as illustrated in the example of Figure 3.3.

(A)  
```
IF
 i IN {0..10} THEN
 (* Some code *)
ELSIF <expression> THEN
 (* Some other code *)
ELSE
 (* Panic!!! *)
END;
```

(B)  
```
IF
 (* Condition *) THEN
 (* Code that goes here *)
ELSE
 (* Not a panic yet! *)
END;
```

(C)  
```
IF
 ELSIF <expression> THEN
 (* Some other code *)
 (* Condition *) THEN
 (* Code that goes here *)
ELSE
 (* Not a panic yet! *)
END;
```

(D)  
```
IF
 <expression> THEN
 (* Some other code *)
ELSIF (* Condition *) THEN
 (* Code that goes here *)
ELSE
 (* Not a panic yet! *)
END;
```

(A) Fragment, containing the If element that is going to be inserted.
(B) Fragment where the insertion is going to take place
(C) Result of the insertion, before re-unparsing is done
(D) Final Result, after unparsing

Figure 3.3 - Example of positional importance for If Elements.

In the special case when the cursor is positioned over complete contents of a fragment, there is no context to take into account, since fragments are context-free. In this case, the cursor can basically be considered to be identical to the fragment (except for information such as system's fragment number, fragment name, etc.). Thus, it is obvious that the cursortype, at this point, is the same as the fragtype. It can thus be said that, within the context of MUPE-2's structured editor, fragtypes are a special case of cursortypes.

A partial list of the cursortypes provided by MUPE-2 is shown in Figure 3.4.
Figure 3.4 - Partial list of cursortypes.

3.2. The Cursor Stack

As stated in the previous section, cursortypes are context sensitive in order to facilitate several of the environment's operations. Every time the cursor is moved to a new position, its cursortype must be updated by examining its contents and its context. Determining the syntactical component of the cursortype is easily done by examining the tag field of the node pointed to by the cursor. However, determining the exact context of the cursor is considerably more difficult. This can involve a complete traversal of the
AST representing the program fragment.

Clearly, if cursor movements are relatively frequent during the use of the environment, it is not practical, nor efficient, to have to traverse the AST representing the whole fragment for every such movement. MUPE-2 uses an alternative approach to solve this problem. The approach is based on the observation that most cursor movements are local. That is, generally they involve only a move to an adjacent node within the AST. This implies that, in the majority of the cases, most of the contextual information gathered in determining the cursortype of the cursor at a certain position within the program fragment can be reused to compute cursortype of the cursor at its new position. Cursortype computation is explained in more details in section 3.4.

The part of the contextual information that can be reused to compute the new cursortype is the hierarchical context. This context, which can also be called lexical, or syntactical, nesting, is the list of the ancestry of the cursor node. The context will generally yield the path to follow in the AST to reach the immediate neighborhood of the new cursor location. Then, only a local context search is needed to complete gathering the contextual information necessary to determine the cursortype. In the environment, the hierarchical context of the cursor is called the cursor trail due to the fact that to get in the position where it is now, the cursor had to move on each and every one of the nodes contained in the path.

The cursor trail is used, by the environment, to provide the hierarchical context of the current cursor for cursortype computations. The context of the cursor is also made available to the user by displaying, with the unparsed program text, the sequence of nested cursors corresponding to the nodes contained in the trail. The resulting display gives cues to the user about such things as the nesting level and context of the current cursor, without cluttering the monitor screen too much. The display of the cursor trail is managed with relative ease: each time the cursor is moved on a structure that is more deeply nested than the previous one, the cursor image, on the screen, shrinks. Thus, it is possible to keep the borders of the old cursor image on the screen every time the cursor moves on a more deeply nested position. An interesting property of the sequence of cursor borders is that it always corresponds to the cursor trail, but can be built, or displayed, without accessing the data structure holding the trail. Also, when moving out of a nesting level, the new cursor will exactly fill the space taken by the cursor trail.
The data structure used to hold the cursor trail is called the cursor stack. Because the level of nesting of structures is not restricted by the environment, nor by the language, it is implemented as a linked list rather than as an array. The elements of the cursor stacks are cursor records. The stack is organized so that the top element is the current cursor and points to the next element in the stack, which represents the parent of the current cursor in the cursor hierarchy.

An interesting point to note is that there is a cursor stack associated with each entity in which independent cursor movement is allowed. That means that there is one cursor stack for each of the programming in the small fragments, and more for the other fragments. For example, in a Statements fragment, there is only one cursor so that only one cursor stack is needed. But in the case of a Procedure fragment, since there are three independent windows, one for each of parameter list, declarations, and statements, there are three independent cursors. Therefore, there are also three cursor stacks for each procedure.

```
CursorStackPtr = POINTER TO CursorStackFrame,
CursorStackFrame = RECORD
  CursorInfo : CursorNode,
  ParentFrame : CursorStackPtr
END;
```

Figure 35 - Record structure of the cursor stack.

### 3.3. Cursor Movements

One of the most frequently used operations on a software document, such as a text file or a program, is the cursor movement operation [Whi82]. While semantically the least significant operations, cursor movements have proved to be indispensable during software development and maintenance phases, since the cursor position in a document designates the software object of interest.

This section is going to discuss the design and implementation of cursor movements. It will consider, in this order, the movements originated at the keyboard via arrow keys, and the movements resulting from the use of a pointing device such as a mouse.
3.3.1. Keyboard Originated Movements

Cursor movements, similar to cursor interpretation, depend on the program view supported by an environment. For example, in Magpie, which supports a textual view of programs, the cursor movements are limited to the movements available to a text editor. In Gandalf, which provides a rigidly structured view of programs, the cursor movements are restricted to those movements permitted in a tree, that is, go to next, previous, parent, or child structure. These movements are often awkward to the users because the screen display shows a textual representation of the program, while the cursor movements on the screen have little or no relation with the sequential nature of text.

On the other hand, environments such as Poe, Pecan, the Cornell Program Synthesizer and Cépage [Mey87], have program views which are partly structured and partly textual. In Poe, Pecan and the Synthesizer, the cursor moves from structure to structure, in a depth-first manner, on the structured constructs of the language supported, and textually everywhere else. These cursor movements are more natural to the users than those of Gandalf and more useful because they reflect both the textual and structural nature of the displayed program.

However, Cépage gives even more importance to the textual view of programs. In Cépage, program constructs are represented as a sequence of windows or cursor positions. For example, an assignment statement is composed of three windows: the designator, the assignment symbol (=) and the expression. The cursor movement rules are simply stated: when moving up, or down, move the cursor to the first window of the previous, or next, line, respectively. This results in an almost purely textual cursor movement scheme. When moving left, or right, move the cursor to the previous, or next window on the same line, respectively. But, cursor movements within expression is purely structural, like those of Gandalf.

MUPE-2 attempts to integrate the structured and textual views of programs with the cursor movements it provides. The philosophy, behind the environment's cursor movements is to provide to the users a set of movements that should, at the same time, both represent adequately the structured nature of program text, and prove to be natural to use.

In contrast to environments that provide a similar program view, the cursor movements provided by MUPE-2 are not restricted to Up and Down as provided by a strict
depth-first traversal of the program tree. The environment provides Up and Down cursor movements which are based on a depth-first traversal of the program tree, but it also offers Left and Right movements which are based on the formatting scheme used by the environment instead.

Since the user moves the cursor by using the arrow keys present on the keyboard, common sense says that the up-arrow (↑) key should correspond to a cursor movement towards the top of the screen. Similarly, the down-arrow (↓), the left-arrow (←), and the right-arrow (→) should correspond, respectively, to a cursor movement towards the bottom, left, and right of the screen respectively.

This common sense approach is used by full-screen text editors and textual program editors, on a character and line basis. But, in MUPE-2, the importance of the structured nature of program text is considered to be very important, so that the approach is implemented on a structure basis. Thus, the left-arrow key is applicable where there is a structure which is textually displayed to the left of the current cursor. The right-arrow key is similarly applicable where there is a structure textually displayed to the right of the current structure.

For example, if the cursor is on an assignment statement, pressing the left-arrow key will move the cursor onto the left hand side of the statement because the rightmost boundary of the cursor moves to the left, giving the impression that the whole cursor may have moved left. Pressing the left-arrow key another time will result in a null movement, since there is nothing displayed to the left of the cursor. Similarly, if right-arrow key is pressed once, it will move the cursor to the right hand side of the assignment, and pressing it for the second time will result in a null movement. It is also possible to move the cursor from the left part to the right part of the assignment, and vice versa, by using, respectively, the right and the left arrow key.

Similarly, the up and down arrow key movements preserve the structured text view of the program by moving the cursor onto structures that are textually displayed above or under the current cursor location. A strictly depth-first traversal of the program tree does provide most of the functionality required for the up and down cursor movements, but it also includes most of the left and right movements. This amalgamation of functionalities makes the simple depth-first traversal inappropriate for the environment.
Figure 3.6 - Example of left and right cursor movements

However, by modifying the traversal, it is possible to achieve cursor movements that respect the up and down movements desired on the display screen. The overall outlook of the modified traversal is still similar to that of the depth-first traversal. The main difference between the two traversals is that in the modified version, structure boundaries that the cursor would usually move on in a depth-first traversal are bypassed, so as to move on structures most likely to need user attention. This means that, as far as possible, the cursor will not move onto a complete control structure of the language when moving up or down, because such structures are not likely to be edited frequently.

For example, pressing the down arrow when the cursor is on a statement that immediately precedes a while loop will move the cursor on the expression of the loop. This is because the expression is more likely to be edited than the while loop. Another example of the behaviour of the vertical cursor movements can be shown for case elements. Because case statements, or case variants of record types, can be very large, it is interesting for the user to be able to quickly search for a given case element. Since case elements can be identified by their label list, when the cursor is on a label list, a vertical movement arrow key (↑ or ↓) will always move the cursor to the label list of an adjacent case element, if there is one. This happens regardless of the number of lines displayed between the two label lists. In the case where there is no adjacent case element, two different actions can be taken, depending on whether the arrow key pressed was the up-arrow or the down-arrow. If the up-arrow was the key being pressed, then the cursor moves onto the case selector expression. But, if it was the down-arrow that had been pressed, then the cursor does not move and a bell is sounded. The reason behind this behaviour of the cursor movements has to do with the number of cursor movements that
the user needs to perform to move back to the original cursor position.

The environment's philosophy, in this case, is that if the user can reach a given location in one cursor movement, he should be able to move back close to the previous location with one more movement. The problem, with the case elements label list occurs when moving down from the last one. The cursor would then move on the first structure following the case statement, but when trying to move up from this structure, there is only one obvious destination for the cursor. The destination is the last statement of the last case element. This is because, apart from the cursor trail, the environment does not retain information about the sequence of structures the cursor has moved on before reaching the current location. For example, moving down from the last statement of the case element would also bring the cursor on the first structure following the case statement. Then, moving up again can only bring the cursor back to the last statement of the last case element because the structure is in the locality of the statement. Thus, if locality is not used, it is impossible to determine whether the cursor should move back on the label list or on the statement. Hence the decision to not allow going down from the last label list solves this problem.

![Diagram showing examples of vertical cursor movements.](image)

Figure 37 - Examples of vertical cursor movements.

It is interesting to note that, even though the environment aims to integrate the textual and structured views of programs, both strictly textual and structured cursor
movements are also provided. Structured cursor movements are provided to permit the
users to move the cursor on the control structures of Modula-2 and to allow fast move-
ment, from structure to structure, through the program text. There are three structured
movements available. Next (<CTRL>-N), Previous (<CTRL>-P), and Out
(<CTRL>-O). These movements correspond respectively, as their names suggest, to
moving the cursor to the next, or previous, structure at the same lexical level, and out a
lexical level, to the parent structure. There is no structured operation provided to move
inside a given structure, as one of the arrow-keys will always achieve this result. Simi-
larly, since it is possible to edit the contents of a cursor textually and to move the cursor
over the complete contents of a fragment, it is therefore possible to edit a complete frag-
ment textually. In this case the cursor movements correspond to the arrow keys found
on most terminals Up or Down a line, and Left or Right a character.

The algorithms for cursor movements in the AST and in the textual representation
of fine grain structures are straightforward in that they are easily understood. For
example, the algorithm to move the text cursor up one line can be stated as

\[
\text{IF line index > 0 THEN decrement line index by one}
\]
\[
\text{ELSE sound the bell}
\]

Similarly, the algorithm to move the cursor from one structure to the next one in the
AST is easily stated as

\[
\text{WITH CurrentNode DO}
\]
\[
\text{IF PointerToNextNode \# NIL THEN}
\]
\[
\text{CurrentNode = PointerToNextNode}
\]
\[
\text{ELSE}
\]
\[
\text{sound the bell}
\]

However, the algorithms to perform cursor movements on structured text, following the
modified depth-first scheme, are neither simple to understand nor easy to implement.

Knowing that the text is structured allows the algorithm to re-use part of the algo-

rithms that perform structured cursor movement. For example, moving down from a
procedure call statement can be reduced to the structured cursor movement. Next in most cases, the structured cursor movement algorithms are sufficient to describe only a small part of the cursor movements that are needed. It is necessary to use a composition of the structured movements with a set of other movements to fully cover the range of cursor movements for structured text.

The extra set of cursor movement algorithms is needed because the structured cursor movements do not allow moving on certain parts of the program text. For example, if the cursor is on a while loop, there is no possible way to move onto the while expression by using one of the three structured movements. The structured movements and the extra movements needed to move inside structures are considered, for the structured text cursor movements, to be primitive operations that will have to be combined in some sequence to achieve the movement desired.

It is not known beforehand whether there is a valid destination for the cursor, when an arrow key is pressed. Therefore, simply executing a sequence of primitive movements with the hope of finding a destination is not safe. This is because there is no way to backtrack to the original cursor location, unless the sequence of primitives is recorded. To solve this problem, the environment uses an algorithm that simulates cursor movements, and at the same time, builds a list of primitives to be executed if a destination is found for the cursor. If no destination is found during the simulation, the algorithm sounds the bell and empties the list of primitives. In the case when a destination is found, the list of primitives is executed.

The destination of a given cursor movement varies depending on the current cursor and on the surrounding structures. For example, if the cursor was on the expression of an assignment statement and the key pressed by the user is the down-arrow, some of the various possible destinations can be:

- the expression of the following assignment statement,
- a complete statement, if the following statement is a procedure call, a return, or and exit statement, and
- the expression of the While loop, For loop, or Case statement following the assignment.

A simplified outline of such an algorithm is shown below in Figure 38. The simplification consists in the omission of all the special cases that can arise in normal
situations. The algorithm also assumes that there will always be a reachable destination statement for the cursor.

```
CurrentNodePtr ← Expression,
REPEAT
   CurrentNodePtr ← Parent(CurrentNodePtr)
   UNTIL CurrentNodePtr' Next ≠ NIL,
   CurrentNodePtr ← CurrentNodePtr'.Next,
   LOOP
CASE CurrentNodePtr' StatementKind OF
   (* from an assignment statement, go onto the expression of the following statements *)
   CaseStatement, WhileStmt, ForStmt, WithStmt, Assignment
      CurrentNodePtr ← CurrentNodePtr'.Expression,
      RETURN (* algorithm successful *)
   (since the following statements do not possess expressions on their first line, go onto their children statements, if any *)
   RepeatStmt
      IF CurrentNodePtr' Children = NIL THEN
         CurrentNodePtr ← CurrentNodePtr'.Expression;
         RETURN
      ELSE
         CurrentNodePtr ← CurrentNodePtr' Children
         (* Go through the loop once more *)
      END |
   LoopStmt
      IF CurrentNodePtr' Children ≠ NIL THEN
         CurrentNodePtr ← CurrentNodePtr'.Children
         (* Go through the loop once more *)
      ELSE
         RETURN (* The loop is the destination *)
      END |
   (The following statements do not support structured movements, so they are the destination. *)
   ProcCall, Exit, Return . RETURN |
   (* The if statement is a little different as the cursor will move onto the expression of the THEN part *)
   IfStmt
      IF CurrentNodePtr'.Children ≠ NIL THEN
         CurrentNodePtr ← CurrentNodePtr'.Children' Expression;
         RETURN
      ELSE
         RETURN (* The if statement is the destination *)
      END
   END
   END (* CASE *)
END; (* LOOP *)
```

Figure 38 - Algorithm Down from assignment expression.
Because of these dependencies, the algorithm that simulates the cursor movement is so complex that it has been divided into a two-dimensional array of simpler algorithms. One index is the arrow-key that has been pressed and the other is the cursortype of the current cursor. There are thus four algorithms defined for each cursortype. Given that the number of cursortypes is fairly large, in the order of seven hundred, the number of algorithms needed to perform cursor movement simulation may seem overly large in respect to their importance. But, in most cases, the exact context of the cursor is not important for the algorithm, so that the majority of the movement simulation algorithms are re-used without modification for a group of related cursortypes. Thus, the real number of algorithms needed is an order of magnitude lower than the product of the number of cursortypes and arrow keys would lead to believe.

3.3.2. Cursor Movements Using the Pointing Device: Long-Jumps

The use of arrow keys to move the cursor to a point of interest to the user is usually adequate when the destination structure is textually near the current cursor location, or when the user is just browsing through the program. But when the user wants to quickly move the cursor to a distant structure, i.e., a long-jump, the use of a pointing device is more appropriate.

Neither an explicit keyboard long-jump operation, nor the use of the pointing device, as a means to move the cursor is unique to MUPE-2. For example, the Cornell Program Synthesizer [Ter81b], while not supporting a pointing device, provides a long-jump operation that allows the user to move faster through the program text than allowed by regular cursor movements. In Pecan [Rei84a], long-jump is implemented using a pointing device, whereas the only cursor movements available in IPSEN [Eng87] are long-jumps, using the pointing device.

Although convenient for the user, the implementation of long-jump via pointing devices causes some problems. In particular, the environment must be able to determine precisely which structure has been pointed to, given only screen coordinates at which the pointing device has been clicked.

This problem is due to the fact that while the unparsable gives a mapping of internal structures to the screen representation, for a given fragment, the inverse mapping is not easily achieved. This is because the unparsable is given the coordinates of the screen area,
or window, which is available for its output, whereas the pointing device can move over the complete screen and, thus, span several fragments, windows, and commands.

An obvious solution to this problem is to keep a map going from the screen display to the internal structures. Unfortunately, such a map, being an array of pointers indexed by pixel coordinates, would be so large; on the order of a few megabytes, as to be impossible to keep in memory for most workstations. When workstations will be able to support gigabytes of main memory, then, possibly, such a map will become a viable alternative solution to the problem. But, with the hardware that is currently available, this solution is not appropriate.

The solution used by MUPE-2 is to treat windows and command buttons as internal structures that are unparsed. When unparsing, the unparsers tags every node in the internal structures with the area occupied on the screen. Then, when the pointing device is used, the mapping to internal structures is performed by a search for a structure that contains the pixel coordinates of the current pointer location.

This search is carried out in some order of the internal structures. For example, all the commands will be searched first, because it is estimated that the majority of the uses of the pointing device will be to select commands. Then, the list of windows will be searched to determine which window, if any, contains the coordinates. Once the window has been found, its sub-windows are searched, and then the fragment's AST is searched for the program structure that contains the coordinates in its display area.

Little can be done to optimize the search through the commands or the windows, since their number is usually relatively small. But program fragments can be of arbitrary size, so as to contain an imposing number of AST nodes. Performing a complete traversal of the AST to find the unique node that contains the pointing device coordinates and no other node is feasible, but far from optimal.

It is interesting to note that at this point: 1) the cursor cannot move outside of the text displayed in the chosen window, and 2) there is one node on the cursor stack that corresponds to the innermost cursor boundary to span the whole window. This means that it is possible to use the information present in the cursor stack to narrow down the search rapidly if the current cursor does not contain the pointer coordinates, check if its parent does. When a node on the cursor stack has been found to contain the coordinates, it identifies the maximum subtree that needs to be searched for the precise structure.
being pointed to. As of this writing, the search algorithm has not been implemented yet, and still another idea to reduce the number of nodes to be searched is being investigated. The idea is, when a node \( \beta \) which contains the coordinates has been found, to search only the subtrees having for root the children of \( \beta \) which are not in the cursor stack. This idea will work because it is already known that any child or grand-child of \( \beta \) that appears in the cursor stack does not contain the pointing device coordinates. It is not known yet if this will prove helpful, as it eliminates only about \( \frac{1}{k} \) of the nodes to be searched, where \( k \) is the fan-out of the AST.

Assuming that the structure referred to by the pointing device is not present on the cursor stack, a long-jump involves a partial re-computation of the cursor stack. Given the structure pointed to and the existing cursor stack, this re-computation involves the same kind of search through the AST as was performed to locate the structure in the first place. An obvious solution to this problem is to treat the location search as a simulation of the cursor movements needed to reach the target structure. In this way, the search routine builds a list of cursor actions to be executed when the search is completed successfully. The actions, which are the same as those used for the simulation of keyboard generated cursor movements, are then executed sequentially. The result of following the actions will be to update the cursor stack without performing a second search.

3.4. Computation of Cursortypes

The computations required to determine the cursortype of an arbitrarily chosen cursor location are not trivial. They involve a traversal of the fragment's AST and the collection of contextual information. Fortunately, cursor movements within the fragment do not move the cursor arbitrarily, so that the computation of cursortype is simplified.

When a fragment, or frame, is first activated, the cursor is positioned over the complete frame, so that the cursortype corresponds to the fragtype. This initial positioning of the cursor is critically important as it provides a known, definitive, cursor location and context for the subsequent cursortype computations. All the cursor movements executed within that frame will modify the cursor location and the cursor type. When the frame becomes inactive again, say at the end of a session, the cursor stack, which contains the hierarchical context of the cursor location and the cursor type, is saved with
the contents of the frame. This is done for two reasons: to restore the cursor to its last location when the user re-activates the frame, and to provide the context and positional knowledge needed to perform cursortype computation.

As stated in the previous section, both arrow-key and long-jump cursor movements are first simulated, generating a sequence of primitive actions needed to perform the actual movements. Each primitive action represents movement along exactly one edge of the AST representing the fragment. Waiting for the completion of the actual movements to happen before performing any cursortype computation may increase the complexity of the computation. This is because executing the complete sequence of actions can substantially modify the context between the source and destination locations of the cursor. Instead, it is possible to perform a simple computation after the execution of each one of the primitive actions, so that the sum of all the computations result in the correct cursortype.

Because cursortype computations have incremental properties and the primitive cursor movement actions correspond to structured movements within the AST, the algorithms for performing the computations can be simplified by adhering to the following set of rules:

- If the action corresponds to moving back to the parent structure, then there is no need to compute the new cursortype since it had been saved on the cursor stack previously. The only function of the algorithm is to pop the cursor stack so that what was the parent frame becomes the current, or top frame, corresponding to the current cursor.

- If the action corresponds to moving onto a child of the current structure, the algorithm should push a new frame on the cursor stack. Then, it should use the cursortype of the initial cursor and the information stored in the AST node representing the current structure to determine the cursortype of the new cursor.

- Finally, if the action corresponds to moving the cursor to a sibling of the current structure, the algorithm should replace the cursortype of the top cursor stack frame with the cursortype of the new cursor location. The computation of the new cursortype uses the cursortype of the parent structure and the information contained in the current AST node.
Chapter 4
Conclusions

This thesis has examined the issues involved in the design and implementation of the program representation and related data structures in the MUPE-2 programming environment. The fundamental point that emerges from this work is that tool requirements should be strongly considered in the design of such structures.

In MUPE-2, the design of a number of program representation structures clearly illustrate this point.

- Type Partitions are the result of a program editor requirement for being able to delete, replace, insert and modify parts of a declaration's type definition such as POINTER TO, SET OF, and ARRAY <index> OF, in a simple manner. The solution was to modify the grammar of Modula-2 so that a type definition consists of a sequence of partitions, instead of a hierarchy of types.

- If and Case Elements are the result of requirements for the program editor and the browser. The editor requires that it should be possible to re-order, delete, and insert elements consisting of a conditional expression, or label, and of a list of statements, within an If or Case statement. It also requires that it should be possible to move such elements between any two If or Case statements, respectively. The browser, on its part, requires that it should be possible to move directly from one such element to the next, or previous. The solution to the browser requirement was to treat both the If and the Case statements as containers of a sequence of If and Case elements respectively, by further modifying the language's grammar. The editor requirements were answered by giving the If and Case elements an abstract syntax their interpretation as either a THEN, ELSIF, ELSE or case part is only dependent on their position within the element sequence.

- The Def-Imp module results from the editor, parser, and static semantic analyzer requirements that no editing operation on an implementation or definition module should introduce inconsistencies in its counterpart. This problem was solved by uniting both modules into one structure that can be viewed by the user as separate
definition and implementation modules. This union makes it possible to immediately reflect in the definition part, a change that occurred in the implementation part of such a structure, thus eliminating possible inconsistencies.

- The Fragment, Fragtypes, Cursor, Cursortypes, and Cursor Stack are the result of the requirement by the program editor that editing operations should be context sensitive and that selection errors from the part of the user should be prevented. This requirement was answered by using the fragment, or the cursor, as the object to be operated on and by defining fragtypes and cursortypes to provide the editor with the context and type of the structure to be operated on. The editor can then compute which are the valid operations and options that the user can select. The cursor stack is used by the editor to determine the hierarchic context of the structure on which the cursor is positioned.

Other environments also show evidence that tools requirements were considered for the design of their program representation. For example, IPSEN has a high-level data structure (graph) that contains all the structural information of a software document and acts as a knowledge base for all the tools applicable to the software document [Lew84]. This fact implies that tools requirements were carefully examined before the design of the data structure, as this data structure is now the only means by which the different tools can access information pertaining to the software document.

However, trying to accommodate all tools requirements with the design of program structures is not always a straightforward exercise. This is because requirements of any two tools may sometimes conflict, or even be mutually exclusive. In these cases, compromises have to be made that may be unsatisfactory to one or both of the tools, but both requirements should be at least partially answered.

For example, one editor requirement is that it should be possible to textually edit expressions. A conflicting requirement from the incremental code generator is that it should not have to parse the constructs for which it is generating code. A compromise was made to accommodate both requirements: an expression is internally kept in tree form, but when the user wants to modify it, it acquires a temporary textual representation as well. This textual representation, once editing is done, is passed to the parser which returns an expression tree for the code generator to work on. This compromise satisfies both tools, but is unsatisfactory due to the fact that it requires two distinct
representations for expressions within the environment.

Another area where compromises have to be made is that of the transfer of complexity between data structures and algorithms, due to the speed and memory size factors. An example of such a compromise exists in the current implementation of MUPE-2. Since cursor movements are well defined, it would be possible to keep the different possible path for such movements hard-wired into the internal data structures through pointers. This would provide for very fast and efficient algorithms to actually perform the movements. But, the cost, in memory usage, of keeping all this information in the data structure is prohibitively expensive. Thus, the complexity was shifted from the data structures to the algorithms. Cursor movements, as implemented in MUPE-2, involve approximately 75 different algorithms that simulate cursor movement, to find whether a destination exists before actually moving the cursor. These 75 algorithms amount to about 300K of code.

The implementation of the environment's program representation consists of approximately 700 lines of type declarations. Because of compiler restrictions, every tree node in the representation is of the same size of 78 bytes. Ignoring the need for program and supporting data structure space and the hardware restrictions on space, this would allow the environment to manipulate fragments consisting of about a million statements, or half a million declarations in the 256 megabytes of virtual memory available to each process on the SUN workstation used to implement MUPE-2. It is to be noted that the current implementation of program representation has yet to be augmented to provide information for incremental code generation and interpretation.
References


[Mad87] N H Madhavji, "Fragtypes A Basis for Programming Environments," *To appear in IEEE Transactions on Software Engineering*


Appendix
MUPE-2 Architecture
Figure (a) - Partial MUPE-2 module decomposition.

Meaning of the node labels

<table>
<thead>
<tr>
<th>Node</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>Data Structures</td>
</tr>
<tr>
<td>GM</td>
<td>General Manager</td>
</tr>
<tr>
<td>BL</td>
<td>Buffer Layer</td>
</tr>
<tr>
<td>ECRS</td>
<td>Edit/Compile/Run System</td>
</tr>
<tr>
<td>ADS</td>
<td>Algorithmic Data Structures</td>
</tr>
<tr>
<td>DDS</td>
<td>Data Data Structures†</td>
</tr>
<tr>
<td>CS</td>
<td>Compilation System</td>
</tr>
<tr>
<td>RTS</td>
<td>Run-Time System</td>
</tr>
<tr>
<td>DES</td>
<td>Data Editing System</td>
</tr>
<tr>
<td>CH</td>
<td>Command Handler</td>
</tr>
<tr>
<td>CT/S</td>
<td>Cursor Type and Stack</td>
</tr>
<tr>
<td>US</td>
<td>Unparsing System</td>
</tr>
<tr>
<td>HOSI</td>
<td>Host Operating System Interface</td>
</tr>
<tr>
<td>SM</td>
<td>Screen Manager</td>
</tr>
<tr>
<td>CL</td>
<td>Communication Layer</td>
</tr>
<tr>
<td>FM</td>
<td>Fragment Manager</td>
</tr>
<tr>
<td>CDS</td>
<td>Common Data Structures</td>
</tr>
<tr>
<td>SPS</td>
<td>Specification Data Structures</td>
</tr>
<tr>
<td>ES</td>
<td>Editing Systems</td>
</tr>
<tr>
<td>AES</td>
<td>Algorithmic Editing System</td>
</tr>
<tr>
<td>SES</td>
<td>Specification Editing System</td>
</tr>
<tr>
<td>CM</td>
<td>Cursor Movements</td>
</tr>
<tr>
<td>EO</td>
<td>Editing Operations</td>
</tr>
</tbody>
</table>

† These data structures are used to represent the contents of data fragments.
MUPE-2 is divided into eight main parts, or subsystems (see Figure (a)). These subsystems cover full range of functional and non-functional requirements for the environment. These subsystems are:

- The Data Structures subsystem contains the declarations of all the data structures that are required by the environment. Such structures include the program representation, the fraglib structures, and the user interface structures.

- The Host Operating System Interface acts as an interface between MUPE-2 and the host operating system. It allows bi-directional file transfers, access to tools such as text editors and mailing systems, and access to the operating system kernel.

- The General Manager is the activity coordinator for MUPE-2. It performs the tasks of a dispatcher upon receiving input from a user, or signals from environment tools, it decides upon the action to take, and calls the appropriate tools.

- The Screen Manager is the user end of MUPE-2. It polls the keyboard and mouse for input, and is in charge of all screen output performed by the environment.

- The Communication Layer is the means through which all tools can communicate between each other. Its task is to provide a standard interface to all the tools.

- The Buffer Layer is where the actual communication routines are implemented. This is necessary because MUPE-2 can be implemented as a monolithic program, as a set of co-routines, or as a set of independent processes.

- The Edit/Compile/Run System contains all the programming tools of the environment. It contains the incremental compiler, the run-time system, and various editors.

- The Fraglib Manager contains tools to store and retrieve fragments in the fragment library. It also provides query and navigation tools.

The decomposition tree shown in Figure (a) is expanded to show only those areas that are relevant to the material contained in this thesis. For example, the fragment header and the fragtypes are defined in the Common Data Structures area of the Data Structures subsystem, and the program representation, the cursor stack and the cursor-types are defined in the Algorithmic Data structures area of this same subsystem.

The fragment headers and the fragtypes will be used by both the General Manager, the Fraglib Manager and the Algorithmic Editing System. The General Manager uses the information provided by the fragment headers and the fragtypes to determine the active fragment and to determine the tool to be called. The Fraglib Manager uses this information to thread the different fragments in the fraglib, and to answer the different possible
queries. The Algorithmic editor allows the user to create new fragments, to delete existing ones, and to modify the fragtype, or header contents, of existing ones.

The program representation structures, contained in the Data Structures subsystem, are only used inside the Edit/Compile/Run System. For example, the Parser, in the Compilation System, reads in text and builds a program tree, from the program structures, as it proceeds. The Interpreter, in the Run-Time System, for its part, executes a fragment tree made out of these program structures.

The Algorithmic Editing System, while using the program representation structures for fragment editing, also uses the cursor related structures to perform syntactic and context checking. The cursor related structures are: the cursor, the cursor stack, and the cursortypes. The cursor is used to select operands for the editing operations, the cursor stack is used to first simulate, and then perform cursor movements, and the cursortypes are used to select legal editing operations and operands.

The program representation structures are also used by the Unparsing System, in the Algorithmic Editing System. The unparsentakes as input a fragment (program) tree and produces formatted Modula-2 text as the output.