RULE BASED DATA STRUCTURE ANIMATION

By

LEE HOU HARVICK

Bachelor of Mechanical Engineering

Tongji University

Shanghai, China

1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 1997

RULE BASED DATA STRUCTURE

ANIMATION

Thesis Approved:

J. Chandler

Thesis Advisor

R. E. Hedrick

Jacques E. LaFrance

Thomas C. Collins

Dean of the Graduate College

PREFACE

There are many types of methods to display Animated Data Structures. Most of the methods are very cumbersome and require an in-depth knowledge of a particular animation language. These animation languages require the developer to think in programming terms instead of the rules that are associated with a data structure. This means that for any learner, whether it is a student, developer, or a researcher, they will be required to spend as much time or more in developing the Data Structure Animation software as they would spend developing the actual software in a more traditional programming languages. The following document presents a simplified Data Structure Animator that is designed to be rule based. This will enable the user to design Data Structure Animation using the more natural rules associated with the Data Structure.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Dr. Chandler for all of his help. I would like to thank Dr. LaFrance for his support and advise which made this thesis possible. I would like to thank Dr. Hedrick for all of his time and consideration.

I would also like to give an extra thanks to Dr. Chandler for not only helping me through my thesis work, but all the extra help he has given me during my graduate studies the past two years.

Also, I would also like to give my special thanks to my husband, Ray Harvick, for his suggestions, help, encouragement, understanding and support throughout the whole process. Also, my sincere thanks to my Parents for their support and encouragement.

Finally, I would like to thank the Department of Computer Science, its faculty and staff, for all the support during these two years of study.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

In examining what Ruled Based Data Structure Animation is, it is important to understand the underlying related terms. It is also important to understand any related subject matter. Data Structures have been animated both statically and dynamically. For purposes of this introduction, we concentrate only on dynamic animation. Most Dynamic Data Structure animation is accomplished either using a traditional programming language (such as Pascal, C, or FORTRAN), or Algorithmic Animators (such as Basil or Zeus). To examine what Data Structure Animation is we will examine related issues, which include Animation, Algorithm, Abstract Data Types, Object Inversion, and Data Structure.

Animation refers to any graphic display of information where the information to be imparted to the viewer is conveyed by an image change [BAE74]. *Post-Simulation Animation* visualizes the input, internal, and output behaviors of a simulation model by using the simulation trace data generated from a completed simulation run [BAL90]. *Simulation-concurrent animation* visualizes the input, internal, and output behaviors of a simulation model as the simulation runs [BAL90].

A *Data Structure* describes the way data are organized in a computer program [VIN84]. A *Data Structure* is just a particular representation of a data object [HIL88]. *Data Structures* are the building blocks of computer algorithms [MAN89].

As previously stated, both traditional programming languages and algorithm animation languages have been used to display data structure animation in the past.

> "By a data structure we mean a table of data including structural relationships." [CON79]

> "In its simplest form, a table might be a linear list of elements, when its relevant structural properties might include the answers to such questions as: Which element is first in the list? Which is last? Which elements precede and follow a given one? How many elements are there in the list?" [KNU73]

> "To structure data effectively, it is essential not only to know techniques but also to know when to apply certain techniques." [MUF82]

> "Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types." [WEI93]

Because of the relationship algorithms have to data structures, algorithm animators have been the traditional method of displaying data structure animation. These animation languages require the developer to think in

programming terms instead of the rules that are associated with a data structure. This means that for any learner, whether it is a student or a researcher, he or she will be required to spend as much time or more in developing the Data Structure Animation Software as they would spend developing the actual data structure in the more traditional programming language. This thesis presents a simplified Data Structure Animator that is designed to be ruled based. This enables the user to design Data Structure Animation using the more natural rules associated with the Data Structure.

Data Structure Animation concentrates primarily on displaying the relationship between data, while algorithm animation concentrates on primarily displaying the relationship of a set of instructions and while Algorithm animation can be used to display a data structure, a data structure animator can only display a subset of algorithms as they relate to the structure and relationships among data. Thus an algorithm that describes the interaction between two nodes of data can be displayed by either an algorithm animator and a data structure animator, an algorithm that does not confine its operations or instructions to a data structure cannot be displayed by a data structure animator. The advantages of a data structure

animator are that it has a greatly reduced scope, and therefore it has less complexity. This reduces the learning curve of the user as well as reduces the amount of work required to display a data structure animation. The disadvantage of a data structure animation is that because it has a limited scope, it is limited only to data structure relationships and cannot display information beyond the scope of the relationships among data.

2. Related Work

The History of Data Structure Animation goes back to the 1960's. The initial animation was done using film. This presented a static visualization of a known Data Structure. This meant that the investigator or instructor could not visualize a sequence of events not depicted in the film.

In 1966, a film produced by Knowlton, "L6: Bell Telephone Laboratories Low Level Linked List Language" demonstrated how an assembly level list processing language works [KNO66].

In three other films, "Sorting Out Sorting" [BAE81], "PQ-trees" [BOO75], and "Hashing Algorithms" by Hopgood, demonstrated the importance of film in describing Data Structure Animation [ARR92]. Each of these films described various static Data Structure Animations.

In the 1970's, research began to focus on static animation from the information available to a system debugger at run-time [ARR92]. These animators displayed the

structures of a program during runtime without the need to change the program itself. These animators showed the results of the operations performed but did not show the operation itself [ARR92].

At the present time, there are many forums that permit the creation of animated Data Structures. These may range from static presentation, such as film [ARR92], to more dynamic methods such as Zeus [BRO92], Tidy Animation's [STA92], and GASP [TAL95]. As was pointed out in "Visualization of Geometric Algorithms," by Ayellet Tal [TAL95], most algorithm and Data Structure Animations have been developed using sophisticated software that was designed for general purpose usage. In "Visualization of Geometric Algorithms," the authors points out the need for software solutions that are more tightly scoped and specific. This would allow the software to be tailored for the specific purpose intended. Tal and his co-author then go on to describe their solution specialized to Geometric Algorithms. It is my intentions to seek out a different specialization in Data Structure Animation.

3. Overview of DSA

3.1 What is DSA

DSA (Data Structure Animation) provides an interactive environment that allows the user to design and run data structures in a visual format.

DSA takes as input a text file, with DSA coding in it. It converts this code into interactive objects. These objects provide the mechanism for all interaction.

3.2 Justification for DSA

While there are many ways to display data structures visually, most presentations are either static in nature or only allow a few select data structures that are hard coded into the software. The few software products that allow dynamic creations of data structures are more general in form. While these animators do a good job of animating algorithms, they are very cumbersome for animating data structures.

In contrast to algorithm animators, DSA focuses only on data structures and therefore is able to simplify the method of presenting graphic displays of data structures. Because DSA focuses on data structures, the user is required to do less coding. In fact, with DSA, the user could actually do less coding than he or she would have to do in a traditional language such as C. This allows the user to focus on the data structure rather than the programming environment.

3.3 Advantages of DSA

In comparison with algorithm animators, DSA has the advantages of focusing on data structures. This means that DSA requires significantly less code to animate data structures than a algorithm animation, since DSA builds in data structure relationships into the environment. DSA assumes that all structures will have one list and many nodes. Each list can have one or more pointers to nodes. Each node can have zero or more pointers to other nodes. The other advantage DSA has over algorithm animators is that it assumes that the screen is divided into regions and that only one node or list can occupy that region. This frees the user from having to calculate the location of a node for visual purposes. Another advantage of DSA is: it places nodes on the screen in their relative positions to the list.

Once again, this frees the user from doing any calculations to determine the location of a node. In DSA, only the list is anchored at any location; all nodes are relative to the list. For example, a list may have a root node with a relative position of "down". This means that the system attempts to place the root node one position down from the list. If the root node also had two nodes with a relative position of "down right" and "down left," then the relative positions of the bottom two nodes are down one and to the left or right from the root, and the root has a relative position of down one. This means the right node's relative position is down two and right one. The left node is down two and left one. By freeing the user from making such calculations, the user has more time to consider the data structure relationship without worrying about peripheral matters.

3.4 Disadvantages of DSA

Because DSA focuses entirely on Data Structure Animation, it is limited in scope. DSA is unable to display any algorithm or data structure that does not consist of one list and one or more node types. DSA cannot show such things as pie charts, bar charts, etc. which algorithm animators can do.

Some of the data structures that a fully operational DSA can display include: Stacks, Linked lists, Queues, Binary trees, AVL trees, etc. In fact, DSA can display almost any structure that is a tree. But DSA would struggle with structures such as a hashing table.

4. DSA Data Structure Development Process

4.1 Creating a DSA Program

To create a DSA program, the developer first must create a text file containing DSA code. After the developer has written his/her program, the developer starts DSA. DSA queries the user for a file name. Once DSA receives the name of the file containing the source code for the program, DSA immediately begins to decode it. During the process of decoding the source file, DSA creates a temporary file using the input name and changing the extension to ".tmp". For example, if the developer opens up "Tree.src", DSA will create a temporary file named "Tree.tmp". If the program was decoded properly, DSA will display an empty child window. Otherwise, it will display a message informing the user that there was an error in the program.

4.2 Load and Running a DSA Program

When the user either starts DSA or request a new source file, DSA displays an opening dialog with a list of files. Once the user selects a file, DSA immediately decodes the source.

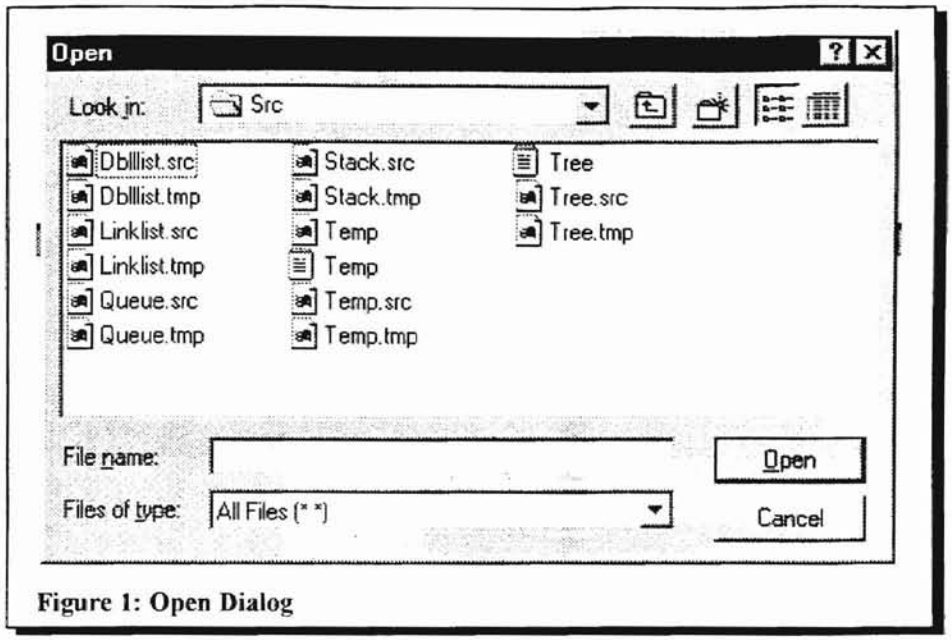


Figure 1: Open Dialog

If the program is loaded successfully without error, DSA will display an empty child window.

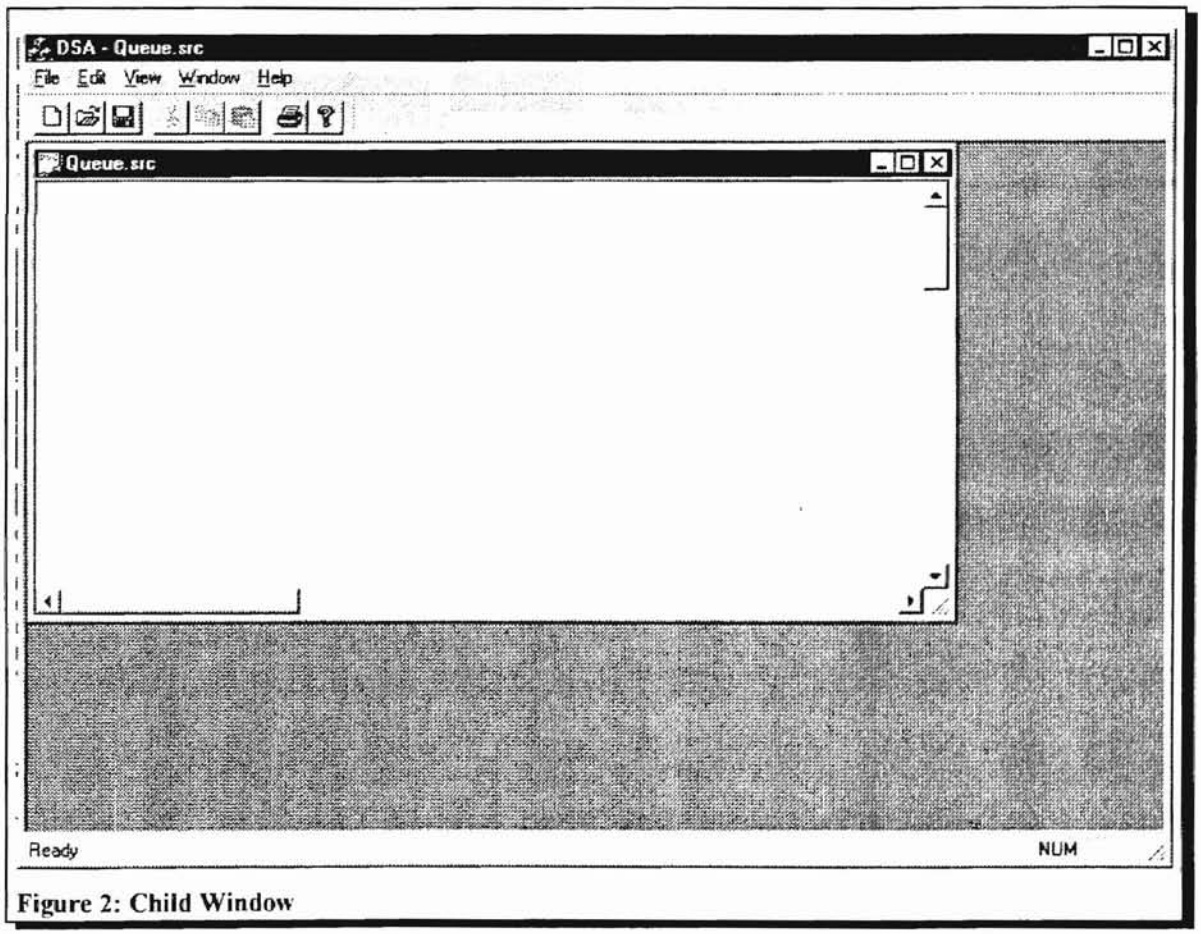


Figure 2: Child Window

Once the user has an empty child window, he/she is ready to begin. The first step in animating a data structure is the creation of a list node. This is done by double clicking anywhere on the blank child window.

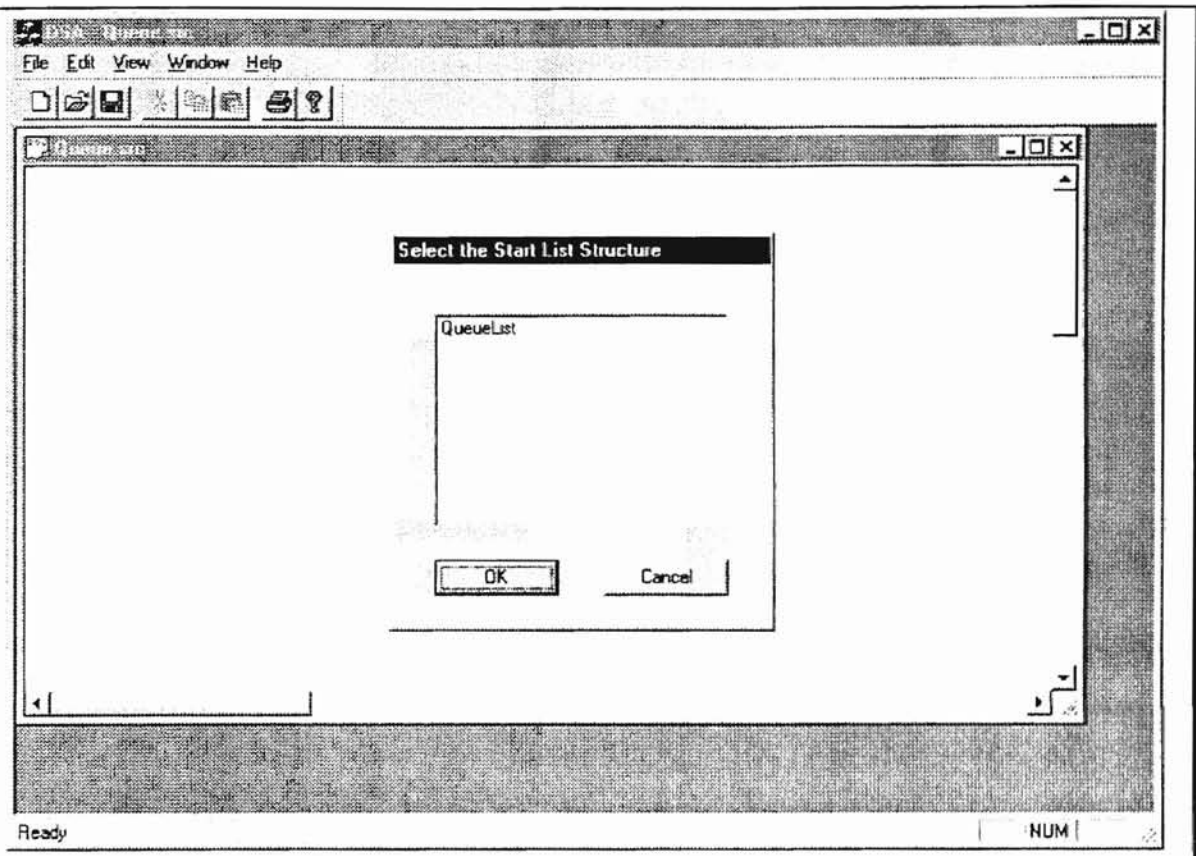


Figure 3: Select a List

After the user has double clicked on a blank portion of the child window, DSA prompts the user to select a list structure. For most DSA source files, there will be only one choice (although it is possible to have more than one list by coding more than one list in the input text file, each list is a separate entity and does not share data). The user clicks on the list of his/her choice and proceeds to the next step.

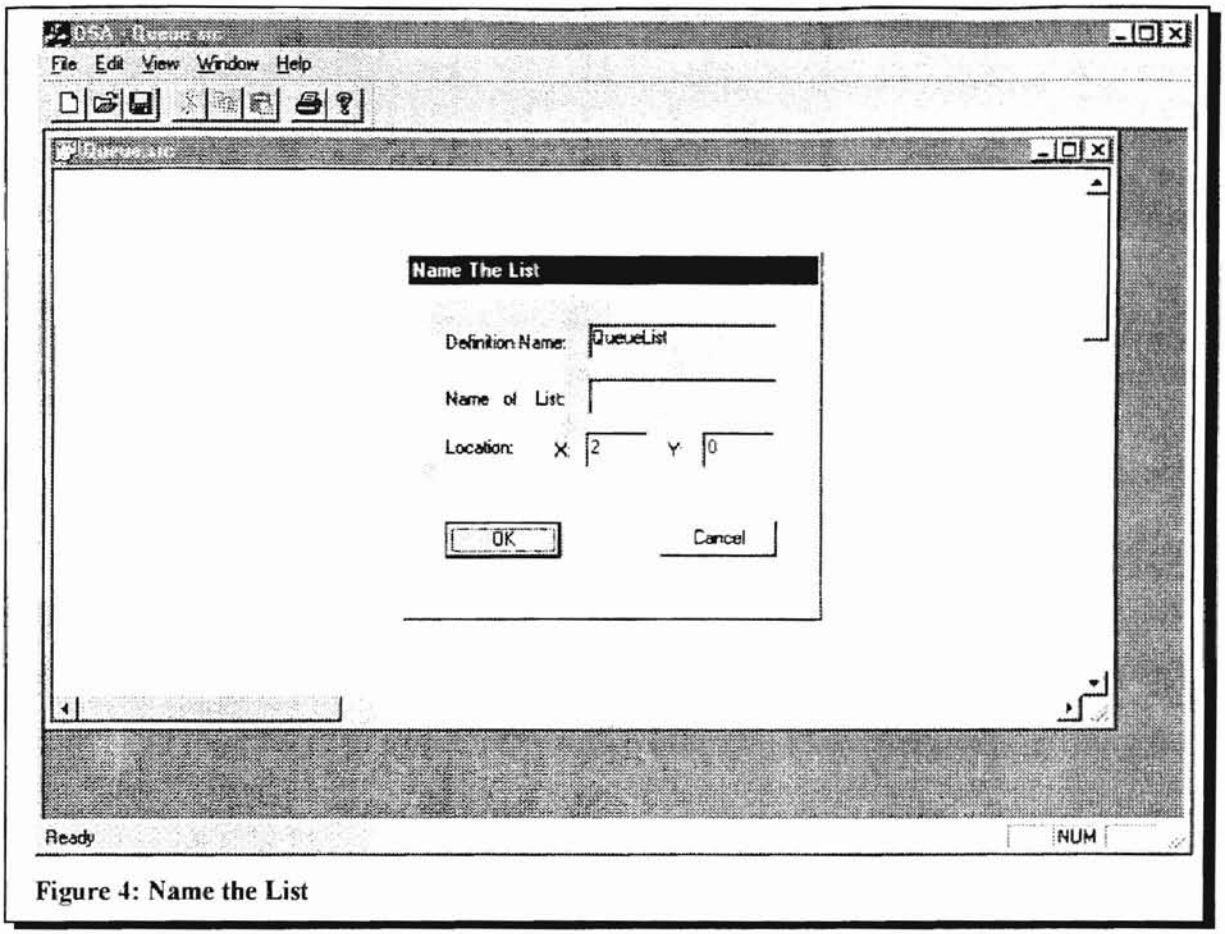


Figure 4: Name the List

After the user has selected a list type, the system queries the user for the name of the list and allow the user to change the physical location of the list. Once the user has entered a name, the system will display an oval, which represents a list, with the name of the list inserted inside the oval.

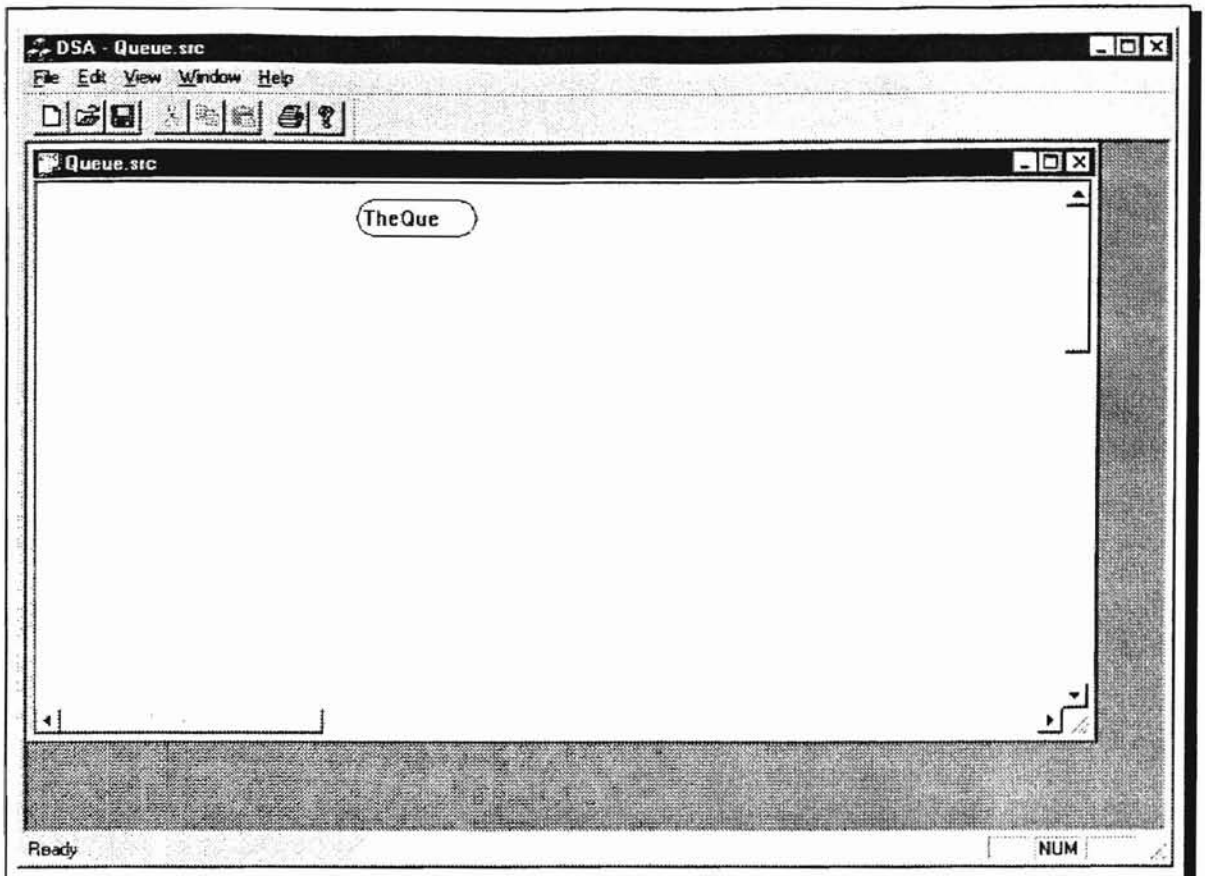


Figure 5: Displayed Queue Node

Once the first list has been entered, the user can either create another list just like the previous one, or the user can run a rule against the list (to insert or remove a node). In order to run a rule (interact with the list), the user double clicks on top of the displayed list (oval).

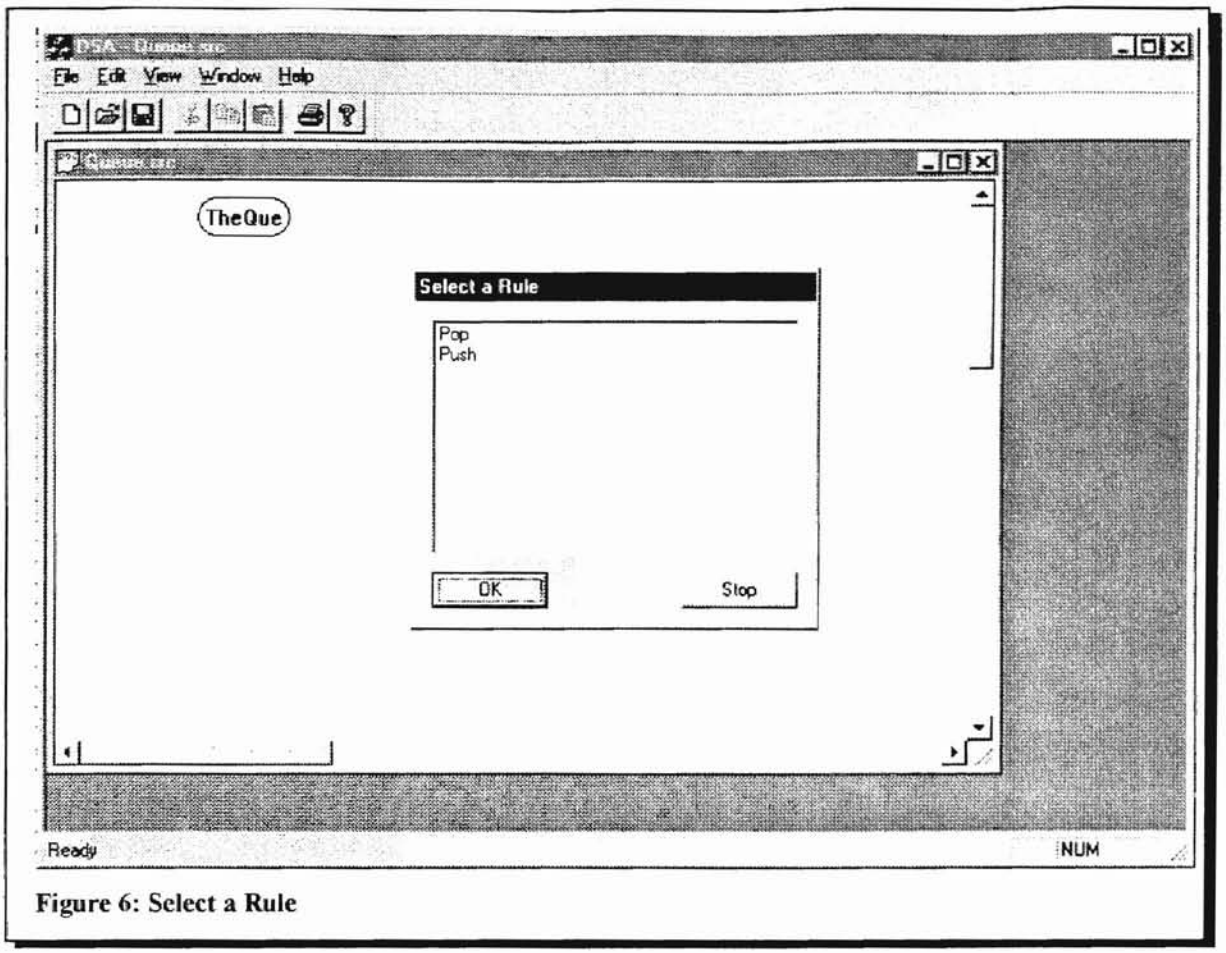


Figure 6: Select a Rule

After the user double clicks on a list (oval), then the system queries the user for the rule to be run. The user is then free to choose the rule to be run. Once the user has started a rule, the rule will be run until either there is a run-time error in the user's program (rule), the rule completes, or the system encounters a request for a new key.

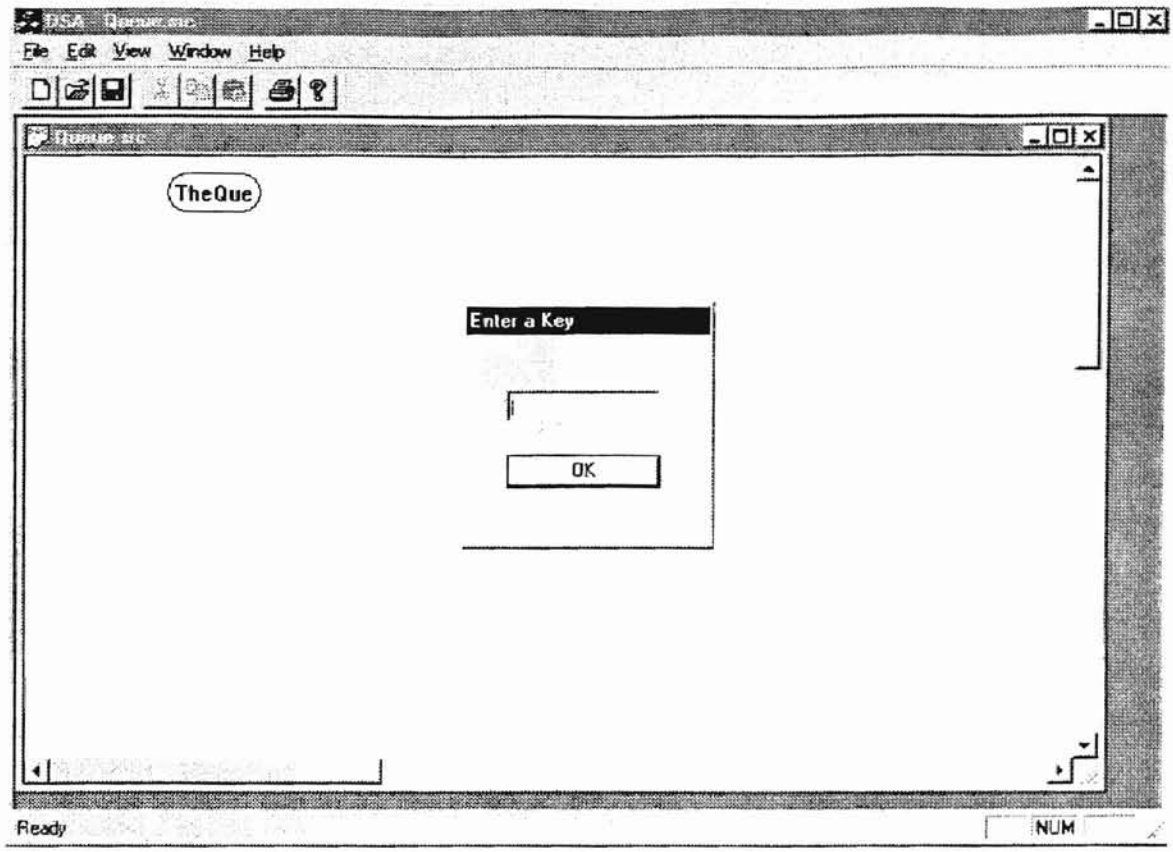


Figure 7: Enter a Key

If during the running of a rule, the system encounters a request for a new key, the system will prompt the user to input a key. After the user enters a key, the system resumes running the rule. After the rule has been run successfully, DSA re-displays the list node and all of its children. On the next pages is the display after one node was entered (first display) and the display after several nodes have been entered.

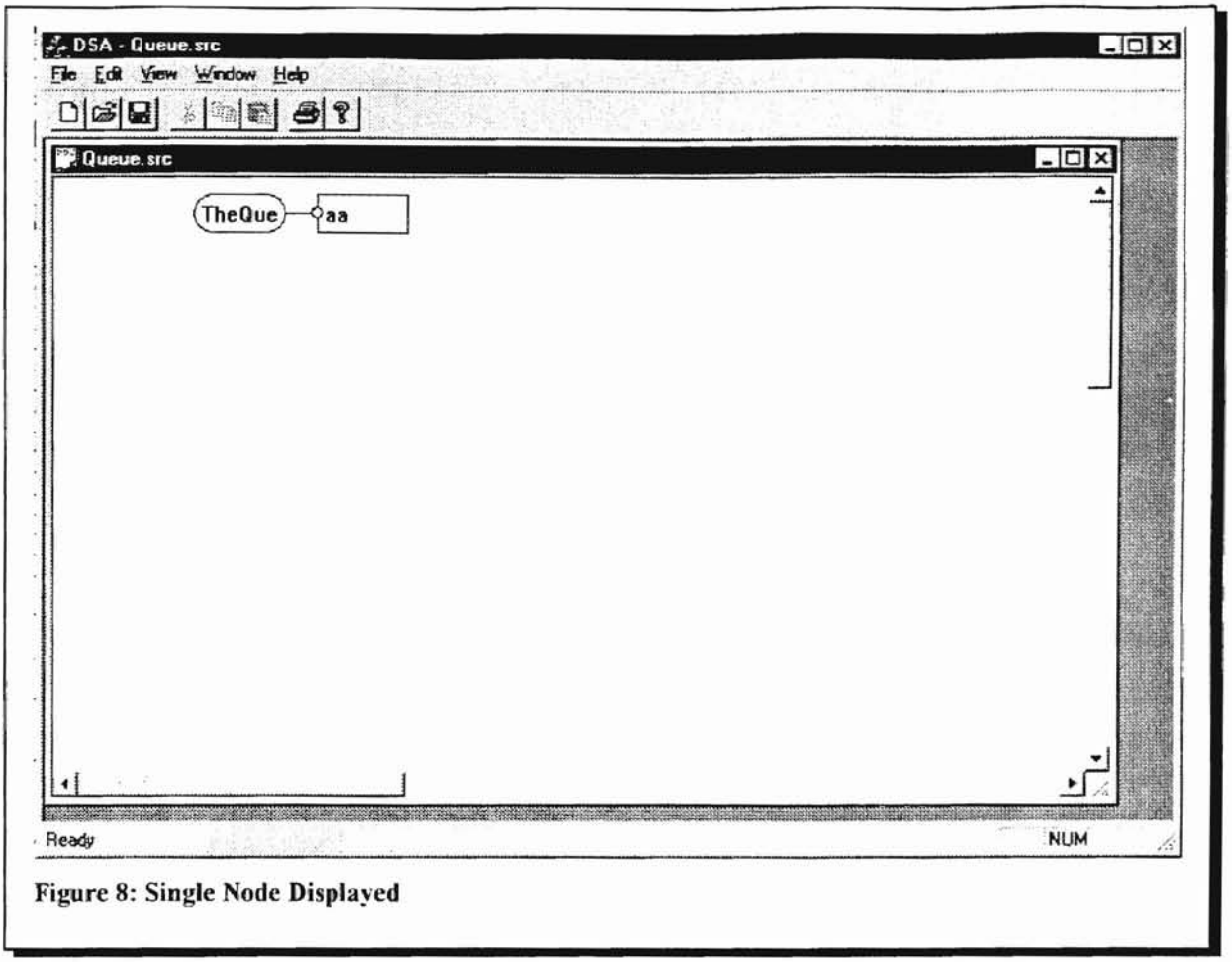


Figure 8: Single Node Displayed

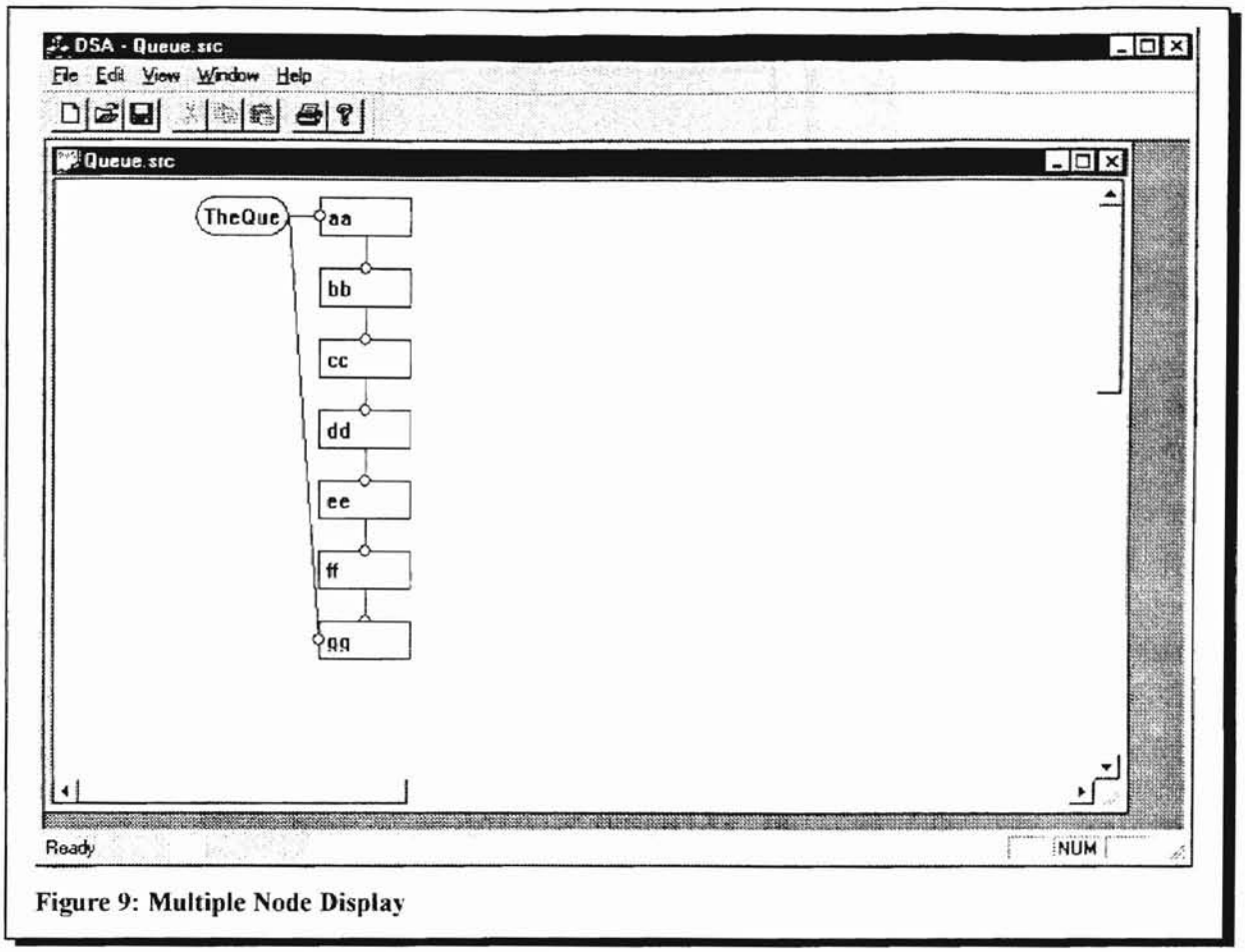


Figure 9: Multiple Node Display

4.3 Debugging a DSA Program

If during the loading of a source file, an error is detected, the system displays a message and halts loading/running the offending source code. Should this occur, the user/developer must change the source code to correct the offending code.

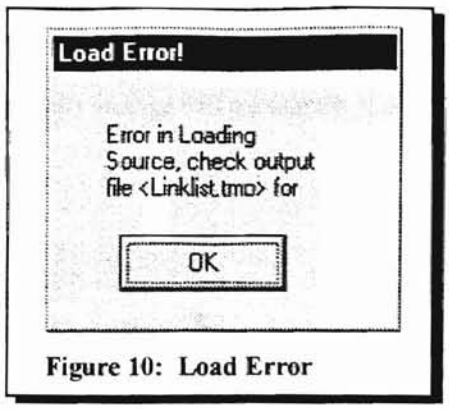


Figure 10: Load Error

Once an error in loading the source code is detected, the developer must fix whatever is wrong with the source program. To help the developer fix bugs in the source code, DSA outputs to a file a copy of the source output along with line numbers, and after each error detected, DSA outputs an error report just below the offending source code.

Example of an Output Source listing with Errors.

```
1::: Single Linked List ::
 2:List: LinkList
 3:  :Node LinkNode Head :Down
 4:  :Insert Insert
 5:  :Remove Remove
 6:End:
 7:
 8::: The Node definition ::
 9:Node: LinkNode
10:  :Node LinkNode Next :Left
11:  :Key  Key
12:End:
13:
14:Rule: Insert
15:  :Node LinkNode NewNode
16:  :New NewNode
17:  :Set @NewNode.Key  <- :NewKey
18:  :Set @NewNode.Next <- Head
19:  :Set Head <- NewNode
20:End:
```

```
 21:
 22:Rule: Remove
 23::: TempNode is used for temporary storage will parsing the link list ::
 24:  :Node LinkNode TempNode
 25:  :Node LinkNode NextNode
 26:  :Node LinkNode PrevNode
 27:  :Set TempNode <- Head
 28:  :Set PrevNode <- :NULL
 29:  :Repeat TempNode != :NULL
Invalid use of Logical Operator inside of <Rule:<Repeat>>
 30:    :If :NewKey = @TempNode.Key
 31:      :If PrevNode = :NULL
 32:        :Set Head <- @TempNode.Next
 33:      :Else
 34:        :Set @PrevNode.Next <- @TempNode.Next
 35:      :Endif
Invalid use of Token :Endif inside of <Rule:<If>>
 36:      :Destroy TempNode
 37:      :End
 38:    :EndIf
Error in defining :Assign detected
 39:    :Set PrevNode <- TempNode
 40:    :Set TempNode <- @TempNode.Next
 41:  :EndRepeat
Error! No Matchin <:Repeat> found for <:EndRepeat>!
 42:End:
Error in defining :Assign detected
Error in defining :Repeat detected
Error in Rule: Remove detected
```

After the source code has been loaded successfully, DSA maintains a log of up to 400 lines of actions performed on the structure. This file has the extension of ".log". For example, if the input file name was Queue.src, the log file would be Queue.log.

```
Loading C:\DSA\DSA\SRC\Queue.src
Creating: TheQue of List Type: QueueList

Running.. Push
:Node QueueNode
:New NewNode
:Set @NewNode.Key <- :NewKey
:Set @NewNode.Next <- :Null
:If Head !=:Null
Branch Else
:Set Head <- NewNode
:Set Tail <- NewNode
End If
```

```
Running.. Push
:Node QueueNode
:New NewNode
:Set @NewNode.Key <- :NewKey
:Set @NewNode.Next <- :Null
:If Head !=:Null
Branch If
:Set @Tail.Next <- NewNode
:Set Tail <- NewNode
End If

Running.. Push
:Node QueueNode
:New NewNode
:Set @NewNode.Key <- :NewKey
:Set @NewNode.Next <- :Null
:If Head !=:Null
Branch If
:Set @Tail.Next <- NewNode
:Set Tail <- NewNode
End If

Running.. Push
:Node QueueNode
:New NewNode
:Set @NewNode.Key <- :NewKey
:Set @NewNode.Next <- :Null
:If Head !=:Null
Branch If
:Set @Tail.Next <- NewNode
:Set Tail <- NewNode
End If

Running.. Push
:Node QueueNode
:New NewNode
:Set @NewNode.Key <- :NewKey
:Set @NewNode.Next <- :Null
:If Head !=:Null
Branch If
:Set @Tail.Next <- NewNode
:Set Tail <- NewNode
End If

Running.. Push
:Node QueueNode
:New NewNode
:Set @NewNode.Key <- :NewKey
:Set @NewNode.Next <- :Null
:If Head !=:Null
Branch If
:Set @Tail.Next <- NewNode
:Set Tail <- NewNode
```

```
End If

Running.. Push
:Node QueueNode
:New NewNode
:Set @NewNode.Key <- :NewKey
:Set @NewNode.Next <- :Null
:If Head !=:Null
Branch If
:Set @Tail.Next <- NewNode
:Set Tail <- NewNode
End If

Running.. Pop
:Node QueueNode
:If Head !=:Null
Branch If
:Set NextNode <- @Head.Next
:Destroy Head
:Set Head <- NextNode
End If

Running.. Pop
:Node QueueNode
:If Head !=:Null
Branch If
:Set NextNode <- @Head.Next
:Destroy Head
:Set Head <- NextNode
End If

Running.. Pop
:Node QueueNode
:If Head !=:Null
Branch If
:Set NextNode <- @Head.Next
:Destroy Head
:Set Head <- NextNode
End If

Running.. Pop
:Node QueueNode
:If Head !=:Null
Branch If
:Set NextNode <- @Head.Next
:Destroy Head
:Set Head <- NextNode
End If

Running.. Pop
:Node QueueNode
:If Head !=:Null
Branch If
:Set NextNode <- @Head.Next
```

```
:Destroy Head
:Set Head <- NextNode
End If

Running.. Pop
:Node QueueNode
:If Head !=:Null
Branch If
:Set NextNode <- @Head.Next
:Destroy Head
:Set Head <- NextNode
End If
```

5. DSA Programming

5.1 Overview of the DSA Programming Language

> "The main purpose of a compiler and of its close cousin, the interpreter, is to translate a program written in a high-level programming language … into a form that the computer can understand in order to execute the program. … An interpreter … translates the source program into an internal form that it can execute." [MAK91]

Based on the above definition, DSA is an interpreted language. DSA reads in source code, translates it into a set of objects which are organized into a set of lists and nodes. DSA run-time objects are a set of lists with nodes that contains lists.

The DSA source code has three distinct types: Lists, Nodes, and Rules. Internally, Lists are made up of a set of nodes and a set of rules. All rules inside of a list are labeled either ":Insert" or ":Remove". These labels do not affect the functionality of a rule; they are for documentation purposes only. Internally, Nodes consists of a set of Nodes and Keys. Rules are comprised of a set of temporary nodes and expressions.

All expressions are objects that share a base class called RuleBase. RuleBase contains a virtual method for every keyword in the DSA language. Many of methods in RuleBase display an error message when called. Every Object expression inherits RuleBase and overrides the methods associated with its type. For example: Set overrides the New method in RuleBase. Since New is only valid with a Set expression, it is the only expression object to override it.

There are two types of stored data, List data and Local data. Any datum (node) that is stored inside of the list is available to the set of rules. In addition, any node that is declared before an expression executes is also available. In order to create a local variable, the ":Node" must be used.

5.2 Keywords

The following is a list of keywords that are used in the DSA language. Some key words are distinguished according to where the ":" is placed. For instance, the ":" in ":Node" is a local reference to a node, while the ":" in "Node:" begins the definition of a node. All definitions (List:, Node:, and Rule: all have ":" as the last character in the key word. All other keywords with ":" have the ":" as the first character. All definitions (List:, Node:, Rule:) all end with an "End:" statement.

Keyword or Symbol	Definition
[A-Z,a-z]+[A-Z,a-z,0-9]*	User Token
=	Equal to
>	Greater than
<	Less than
!=	Not Equal
>=	Greater or Equal to
<=	Less or Equal to
:Insert	Insert rule
:Remove	Remove rule
:Node	Node <Node Data Type>
:Key	Key <Key Data Type>
:Order	Order <Reserved, not used>
:New	New <Create Memory>
:Destroy	Destroy <Destroy Memory>
:Msg	Msg <Display a Message>
:Set	Set <Set a Node or Key>
<-	Assign <Used with set (like "=")>
:If	If
:Then	Then <Reserved, not used>
:Else	Else
:EndIf	End If Statement
:NewKey	New Key <Query user for new key>
:While	While
:Repeat	Repeat
:EndRepeat	End Repeat
:Null	Null <Set Data to Null>
:End	End <Stop running the Rule>
:UpDown	<Reserved, not used>
:DownUp	<Reserved, not used>
:LeftRight	<Reserved, not used>
:RightLeft	<Reserved, not used>
:Ignore	<Ignore Positioning>
:Left	Position Node to the Left
:Right	Position Node to the Right
:Down	Position Node Down
:Up	Position Node Up
:UpLeft	Position Node Up and to the Left
:UpRight	Position Node Up and to the Right
:DownLeft	Position Node Down and to the Left
:DownRight	Position Node Down and Right
::	Start/End Comments
List:	Start definition of a List
Node:	Start definition of a Node
End:	End the definition List,Node,Rule>
Rule:	Define a Rule
:Free	remove reference link of a node
@	Redirect Data Pointer
&	Set Pointer to Reference

5.3 List Definitions

The List: statement defines a list.

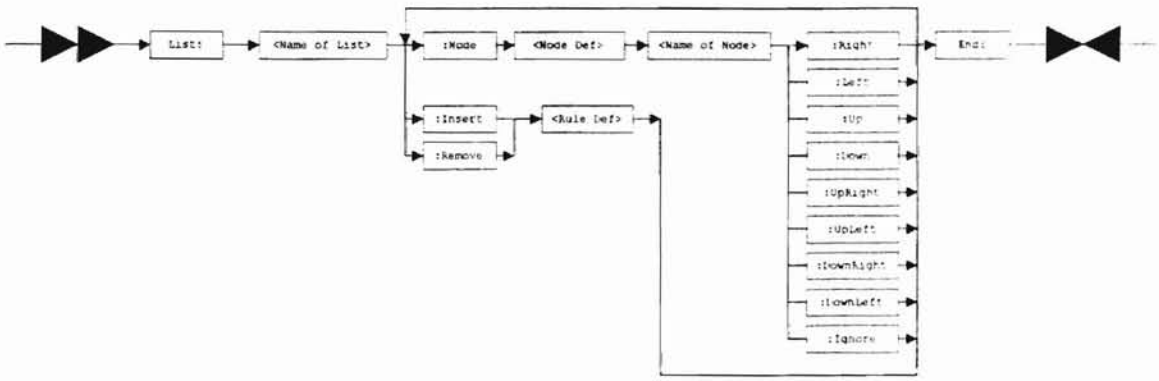


Figure 11: List Definition Diagram

The <Name of the List> can be any name one wishes to give it, including characters, numbers, and symbols, as long it is not a reserved word.

The <Rule Def> can be any name, but at run time there must be a corresponding rule definition or an error occurs. It does not matter whether the rule definition occurs before or after the List:.

The <Node Def> can be any name, but at run time, there must be a corresponding node definition or an error occurs.

It does not matter whether the node definition occurs before or after the List:.

5.4 Node Definitions

The Node: statement defines a node.

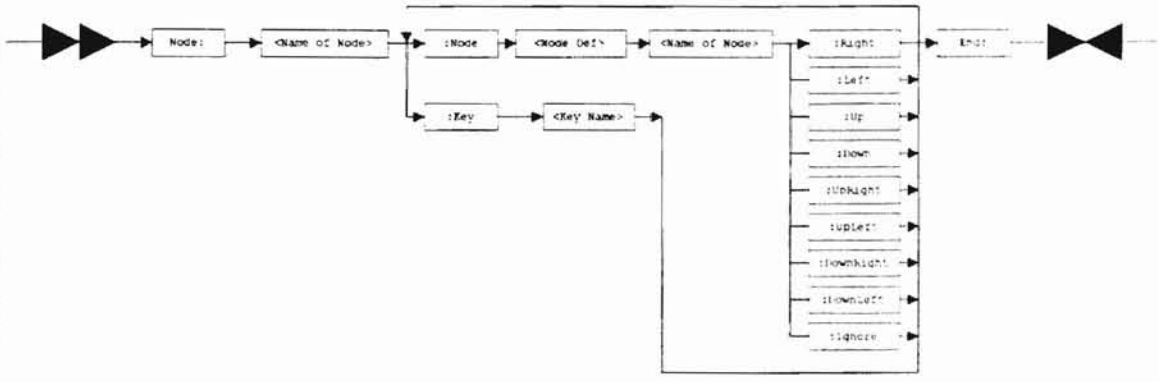


Figure 12: Node Definition Diagram

The <Name of node> can be any name one wishes to give it, including characters, numbers, and symbols, as long it is not a reserved word.

The <Key Name> can be any name one wishes to give it, including characters, numbers, and symbols, as long it is not a reserved word.

Every ":Node" inside of "Node:" must have a relative position key word such as ":Right". This tells DSA that the relative graphical display position of the node link is to

the right of the list. While DSA attempts to place the node one region to the right, it may alter its positioning depending upon other elements (Nodes or Lists) or to give a smoother look to the overall display. For instance, in a tree, DSA may push nodes under the root in both directions in order to reduce the bell curve look.

Each Node: can have as many :Node's or :Key's as is desired.

5.5 Rule Definitions

The Rule: statement defines a rule.

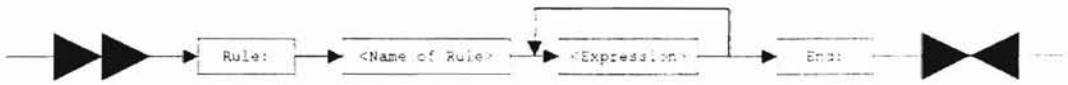


Figure 13: Rule Definition Diagram

The <Name of rule> can be any name one wishes to give it, including characters, numbers, and symbols, as long it is not a reserved word.

Each rule is made up of one or more expressions.

5.6 *Expressions, Conditions, and Statements*

5.6.1 Destroy Expression

The :Destroy statement deletes the memory that to which the <name of node> points.

Figure 14: Destroy Expression Diagram

5.6.2 Message Expression

The :Msg statement displays any message contained between two quote characters.

Figure 15: Message Expression Diagram

5.6.3 Set Expression

The :Set statement sets the value of a node or key.

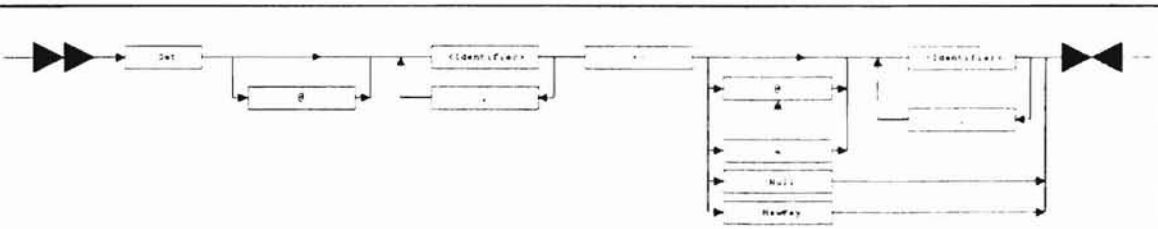


Figure 16: Set Expression Diagram

Each :Set must be followed by an identifier followed by the assignment operator "<-". The Assignment operator is then followed by a second identifier or the keywords ":Null", ":NewKey", or ":Free". The identifier may either be direct or indirect. Any identifier containing a "@" as its first character (or second if the first character is a "&") is an indirect address and must take on the form <identifier>.<identfier>, where the first <identifier> is any node (either local or in a list) and the second <identifier> is either a node or a key that is in the definition of the first <identifier>. If there is a third <identifier> it must be contained in the second, and so forth. Any identifier that begins with the character "&" is a reference. Whenever a reference is used to set an identifier, any future reference to that identifier would be the same as though it accessed the original identifier. For example:

```
:Set x <- &y
:Set x <- :New
```

In this example, y is given new memory. However, we had the following example:

```
:Set x <- y
:Set x <- :New
```

Then y still contains whatever value it had, but x receives new memory. Once a identifier has been set to a reference, it remains a reference until it either is out of scope or it is set to ":Free".

5.6.4 If Expression

The :If statement is a conditional branch.

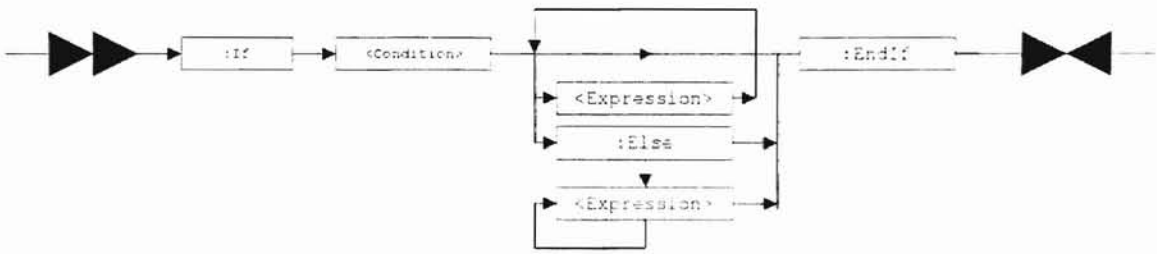


Figure 17: If Expression Diagram

The :If statement will cause the <condition> to be examined, and if true, all expressions listed before the :Else statement are executed. Otherwise, all expressions after the :Else are executed.

5.6.5 Repeat Expression

The :Repeat statement is a conditional loop.

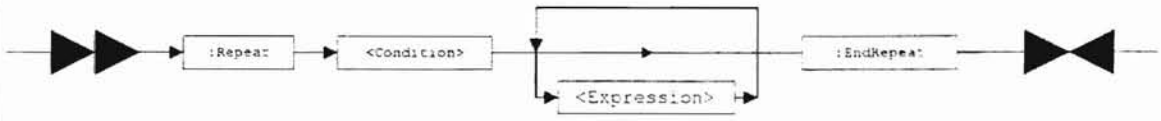


Figure 18: Repeat Expression Diagram

The :Repeat statement causes the <condition> to be examined, and if true, all expressions are executed and then recycled to the beginning and the <condition> is re-examined. Once the condition is false, the system proceeds to the expression after the :EndRepeat.

5.6.6 Conditions

Conditions are used in both the :If and the :Repeat expression.

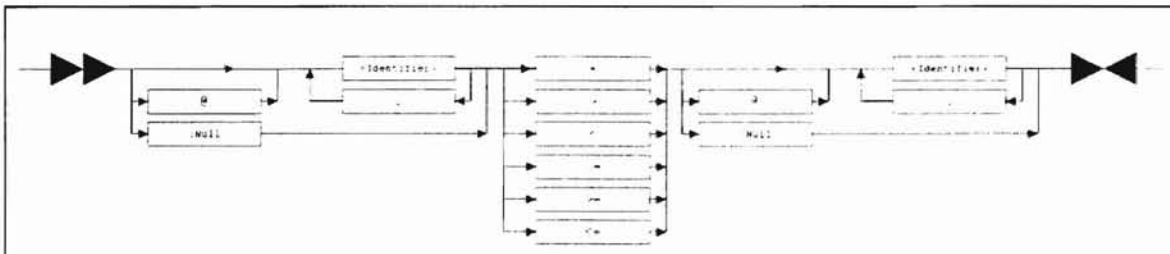


Figure 19: Conditional Diagram

The condition causes the identifier to be on both the left and right side, and based on the operation (<,>,=,!=,>=,<=) returns either a true or a false. Based on these results, the :If or :Repeat flow is determined.

5.7 Example Source code

5.7.1 Queue.src

```
List: QueueList
  :Node QueueNode Head :Right
  :Node QueueNode Tail :Ignore
  :Insert Push
  :Remove Pop
End:

Node: QueueNode
  :Node QueueNode Next :Down
  :Key  Key
End:

Rule: Push
  :Node QueueNode NewNode
  :New NewNode
  :Set @NewNode.Key <- :NewKey
  :Set @NewNode.Next <- :Null
  :If Head != :Null
    :Set @Tail.Next <- NewNode
    :Set Tail <- NewNode
  :Else
    :Set Head <- NewNode
    :Set Tail <- NewNode
  :EndIf
End:

Rule: Pop
  :Node QueueNode NextNode
  :If  Head != :Null
    :Set NextNode <- @Head.Next
    :Destroy Head
    :Set Head <- NextNode
  :Else
    :Msg "No Nodes in List"
  :EndIf
End:
```

5.7.2 Stack.src

```
List: StackList
  :Node StackNode Top :Down
  :Insert Push
  :Remove Pop
End:

Node: StackNode
  :Node StackNode Next :Down
  :Key  Key
End:

Rule: Push
  :Node StackNode NewNode
  :New NewNode
  :Set @NewNode.Key <- :NewKey
  :If Head != :Null
    :Set @NewNode.Next <- Top
    :Set Top <- NewNode
  :Else
    :Set @NewNode.Next <- :Null
    :Set Top <- NewNode
  :EndIf
End:

Rule: Pop
  :Node StackNode NextNode
  :If  Top != :Null
    :Set NextNode <- @Top.Next
    :Destroy Top
    :Set Top <- NextNode
  :Else
    :Msg "No Nodes in Stack"
  :EndIf
End:
```

5.7.3 □Tree.src

```
List: TreeList
  :Node TreeNode Top :Down
  :Insert Insert
  :Remove Remove
End:

Node: TreeNode
  :Key  Key
  :Node TreeNode LeftChild  :DownLeft
  :Node TreeNode RightChild :DownRight
End:

Rule: Insert
  :Node TreeNode NewNode
```

```
  :Node TreeNode TopNode
  :New NewNode
  :Set @NewNode.Key <- :NewKey
  :Set @NewNode.LeftChild <- :Null
  :Set @NewNode.RightChild <- :Null
  :If Top = :Null
    :Set Top <- NewNode
  :Else
    :Set TopNode <- &Top
    :Repeat TopNode != :Null
      :If NewNode > TopNode
        :If @TopNode.RightChild = :Null
          :Set @TopNode.RightChild <- NewNode
          :Set TopNode <- :Free
        :Else
          :Set TopNode <- &@TopNode.RightChild
        :EndIf
      :Else
        :If @TopNode.LeftChild = :Null
          :Set @TopNode.LeftChild <- NewNode
          :Set TopNode <- :Free
        :Else
          :Set TopNode <- &@TopNode.LeftChild
        :EndIf
      :EndIf
    :EndRepeat
  :EndIf
End:

Rule: Remove
  :Node TreeNode NewNode
  :Node TreeNode TopNode
  :Node TreeNode NodeTop
  :Node TreeNode SwapNode
  :New NewNode
  :Set @NewNode.Key <- :NewKey
  :Set @NewNode.LeftChild <- :Null
  :Set @NewNode.RightChild <- :Null
  :Set TopNode <- &Top
  :Set NodeTop <- :Null

  ::**********************************::
  :: Find the Node to be replaced     ::
  ::**********************************::

  :Repeat TopNode != :Null
    :If NewNode = TopNode
      :Set NodeTop <- &TopNode
      :Set TopNode <- :Free
    :Else
      :If NewNode > TopNode
        :If @TopNode.RightChild = :Null
          :Set TopNode <- :Free
        :Else
```

```
      :Set TopNode <- &@TopNode.RightChild
    :EndIf
   :Else
    :If @TopNode.LeftChild = :Null
     :Set TopNode <- :Free
    :Else
     :Set TopNode <- &@TopNode.LeftChild
    :EndIf
   :EndIf
  :EndIf
:EndRepeat

::*****************************::
:: Find Node to be swapped in  ::
::*****************************::


:If NodeTop != :Null
 :If @NodeTop.RightChild = :Null
  :If @NodeTop.LeftChild = :Null
   :Set NodeTop <- :Null
  :Else
   :Set NodeTop <- @NodeTop.LeftChild
  :EndIf
 :Else
  :If @NodeTop.LeftChild = :Null
   :Set NodeTop <- @NodeTop.RightChild
  :Else
   :Set TopNode <- &@NodeTop.LeftChild
   :Set SwapNode <- :Null
   :Repeat TopNode != :Null
    :If  @TopNode.RightChild = :Null
     :Set SwapNode <- TopNode
     :If @TopNode.LeftChild = :Null
      :Set TopNode <- @TopNode.LeftChild
      :Set TopNode <- :Free
     :Else
      :Set TopNode <- :Null
      :Set TopNode <- :Free
     :EndIf
    :Else
     :Set TopNode <- @TopNode.RightChild
    :EndIf
   :EndRepeat
   :If SwapNode != :Null
    :Set @SwapNode.RightChild <- @NodeTop.RightChild
    :Set @SwapNode.LeftChild  <- @NodeTop.LeftChild
    :Set NodeTop <- SwapNode
   :Else
    :Msg "Unkown Error, SwapNode is Null???"
   :EndIf
  :EndIf
 :EndIf
:Else
```

```
    :Msg "Unable to find Node in Tree"
  :EndIf
End:
```

5.7.4 LinkList.src

```
:: Single Linked List ::
List: LinkList
  :Node LinkNode Head :Down
  :Insert Insert
  :Remove Remove
End:

:: The Node definition ::
Node: LinkNode
  :Node LinkNode Next :Left
  :Key  Key
End:

Rule: Insert
  :Node LinkNode NewNode
  :New NewNode
  :Set @NewNode.Key  <- :NewKey
  :Set @NewNode.Next <- Head
  :Set Head <- NewNode
End:

Rule: Remove
:: TempNode is used for temporary storage will parsing the link list ::
  :Node LinkNode TempNode
  :Node LinkNode NextNode
  :Node LinkNode PrevNode
  :Set TempNode <- Head
  :Set PrevNode <- :NULL
  :Repeat TempNode != :NULL
    :If :NewKey = @TempNode.Key
      :If PrevNode = :NULL
        :Set Head <- @TempNode.Next
      :Else
        :Set @PrevNode.Next <- @TempNode.Next
      :Endif
      :Destroy TempNode
      :End
    :EndIf
    :Set PrevNode <- TempNode
    :Set TempNode <- @TempNode.Next
  :EndRepeat
End:
```

5.8 Example Data Structure Output Images

5.8.1 Queue Structure Output image

In the figure below, the key values aa,bb,cc, and dd were entered in sequence, with the following results:

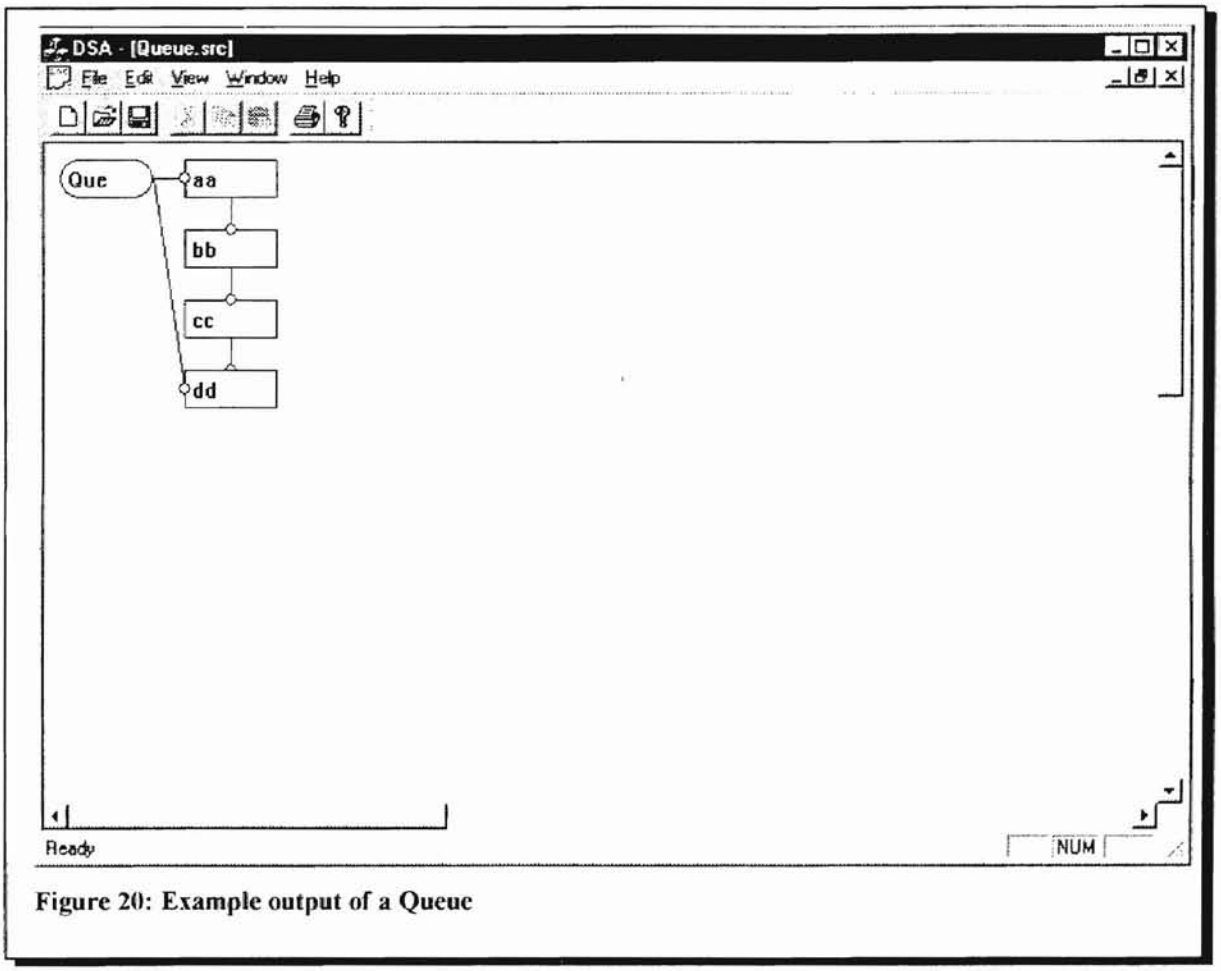


Figure 20: Example output of a Queue

5.8.2 Stack Structure Output image

In the figure below, the key values aa, bb, and cc were entered in sequence, with the following results:

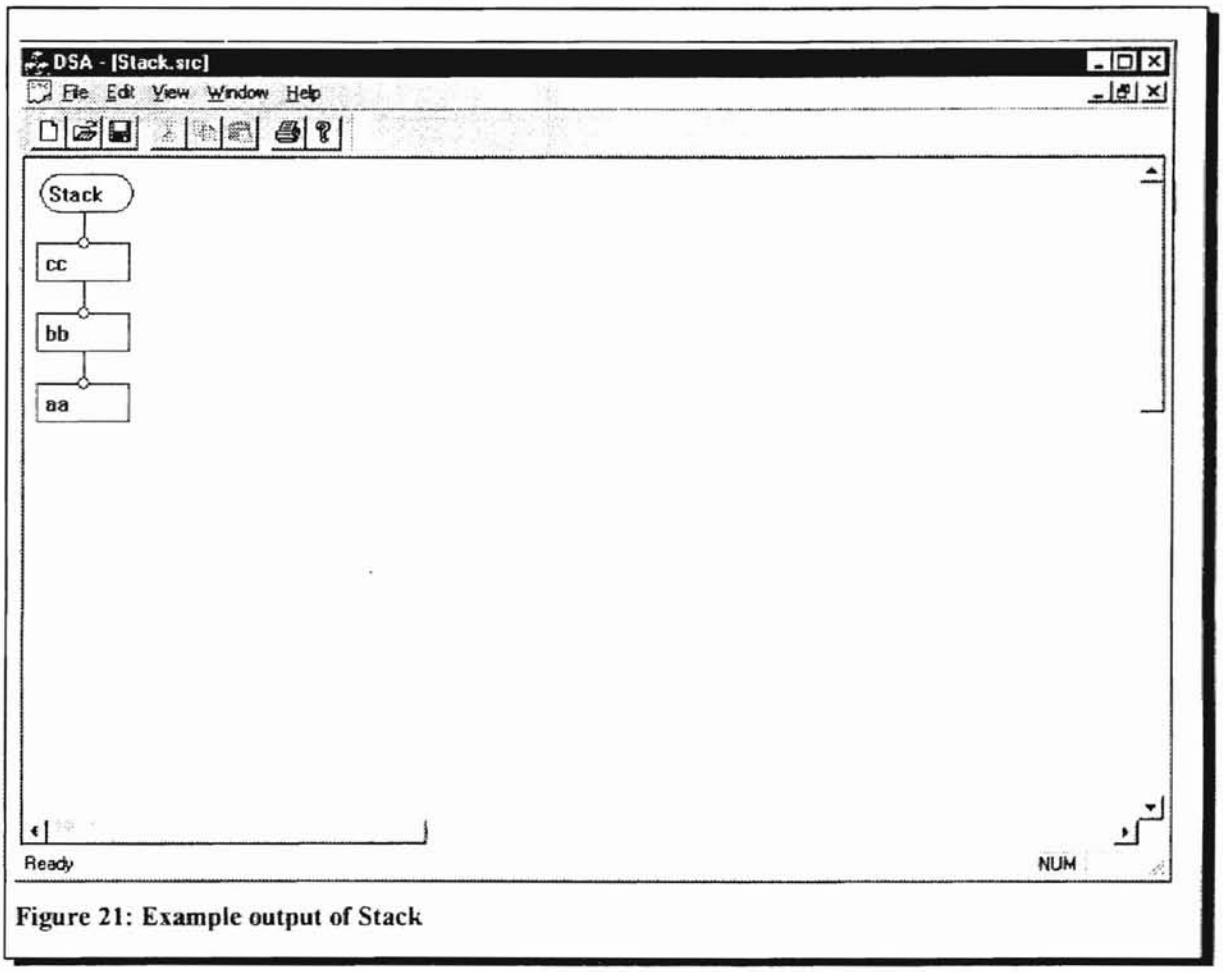


Figure 21: Example output of Stack

5.8.3 Binary Tree Structure Output image

In the figure below, the key values mm, gg, ss, bb, ii, and tt were entered in sequence, with the following results:

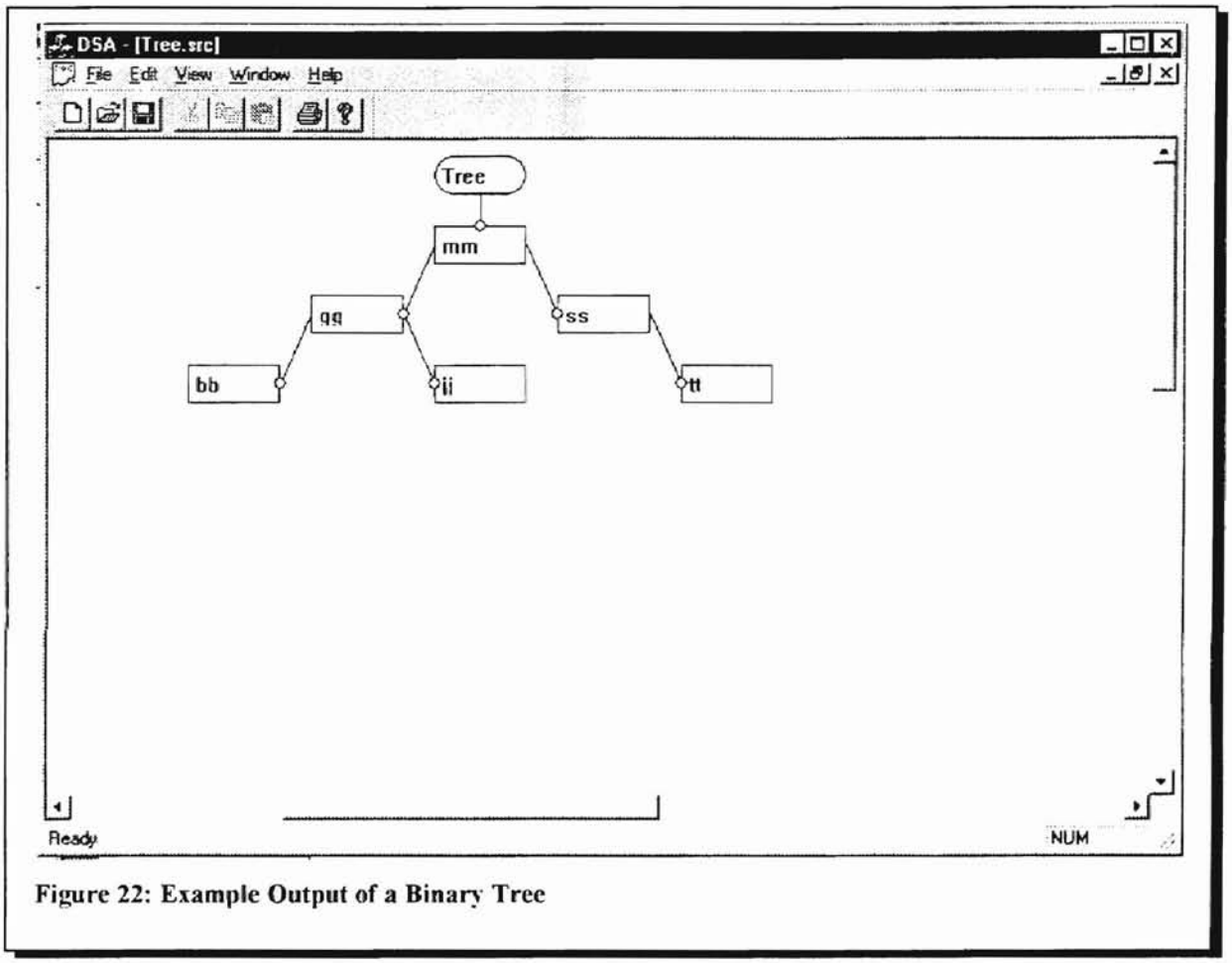


Figure 22: Example Output of a Binary Tree

6. DSA Architecture

6.1 Overview of Architecture

DSA is based internally on object oriented technology. DSA also utilizes Microsofts Foundation classes for all Windows interfaces. DSA utilizes two base classes extensively (LinkList and LinkNode). These two classes form the bases of almost all functionality within DSA. This is done by either using the data elements inside of LinkNode (FieldName and FieldData) or by inheriting either LinkList or LinkNode, or a combination of both (usually a combination of both). Often objects are stored inside a linked list as a LinkNode object and then recast to the original class after they are retrieved from the list.

6.2 Class/Object relationships

6.2.1 Principal Class/Object Relationships

Many of the objects in the chart below are instances of MFC classes or classes that are built upon and inherited from MFC architecture. In the chart below, all rectangles represent classes, all ovals represent objects, and the one hexagon represents the main function and the starting address.

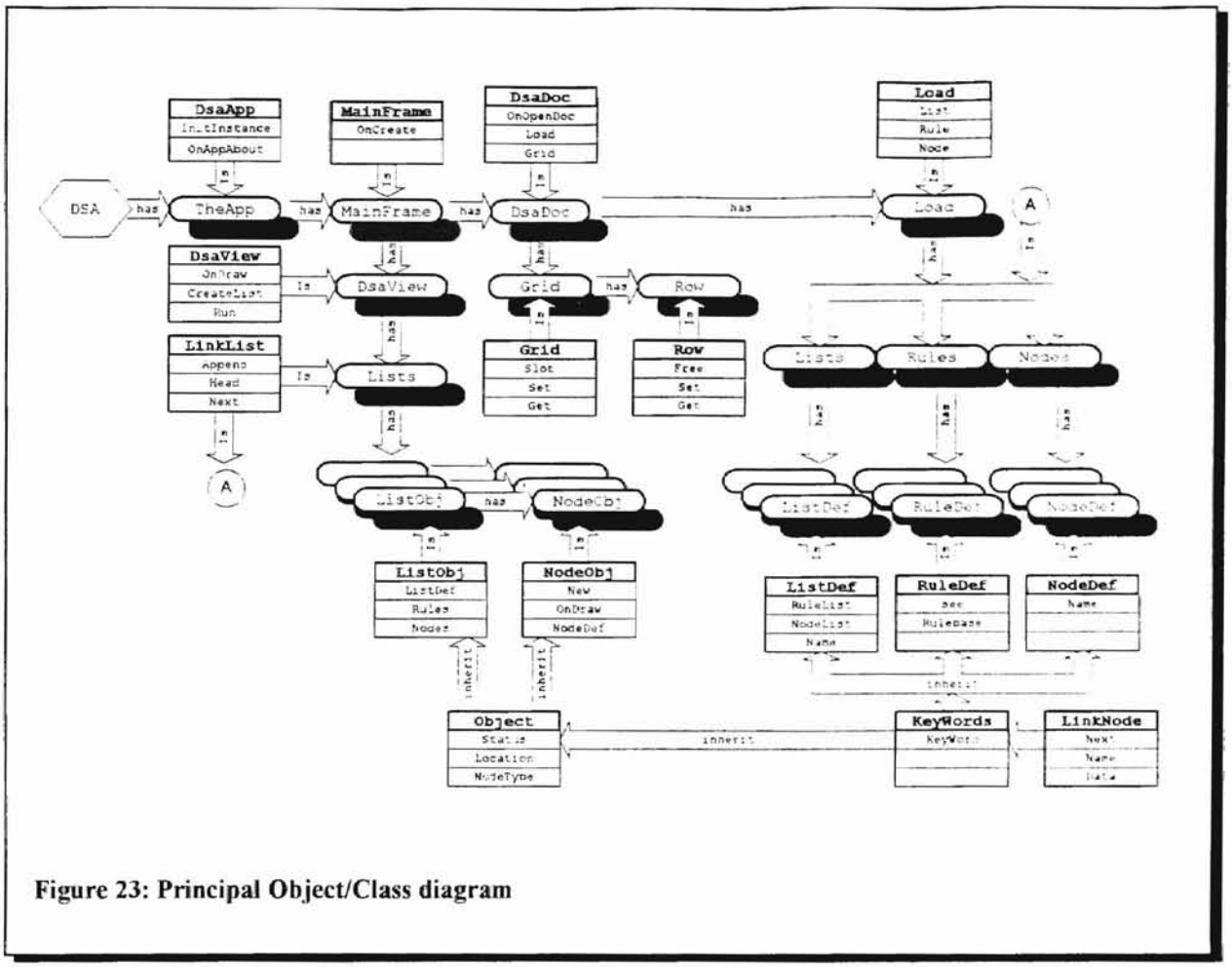


Figure 23: Principal Object/Class diagram

All following classes that are used to instantiate objects are described in detail. The inheritances involved in the classes above. The classes, ListDef, RuleDef, NodeDef, and Object all inherit Keywords. Keywords is used to identify a word and determine whether it is a keyword and if so, what kind.

6.2.2 Rule Class/Object Relationships

In the chart below, the hierarchy of all expressions and rules is shown. All expressions and RuleDef inherit RuleBase. RuleBase contains two very important methods. Both methods are named Run, but have a different signatures (parameters). The first Run is used to scan the input file. Once a token is read from the input file, it is examined using Keywords for its type. Once its type is determined, either an error is produced and printed to the ".tmp" file or an object of the correct type is created. Once the object is created, it is given access to the input source, and continues to read the source until it either loses scope (as with a :If finding a :EndIf) or it returns control to its calling object if an error occurs.

The second run is for running the actual rules. Each RuleDef is a LinkNode that contains a LinkList of expressions. Rule processes each expression (using the "Run" method inside of RuleBase). Each Method inside of RuleBase is a virtual method. As each expression is processed, RuleBase calls the appropriate virtual method. That method may be from the RuleBase class or an overloaded method from another class. For instance, the key word

":New" is invalid inside of most expressions and therefore RuleBase contains a virtual method to display an error when it encounters a ":New". If the object that inherited RuleBase is ":Set", then when RuleBase calls the virtual method for ":New", it will execute the overload method inside of ":Set" instead of RuleBase.

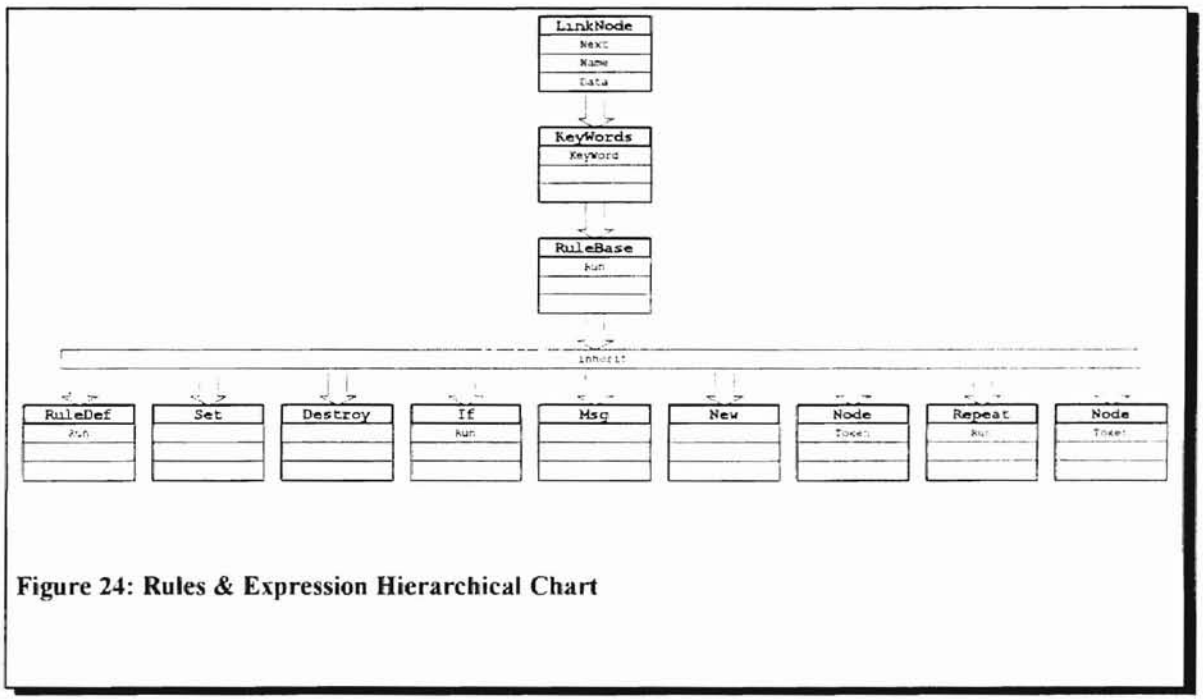


Figure 24: Rules & Expression Hierarchical Chart

The above classes are described in detail in the ensuing pages.

6.3 Example Rule Object Relationship Chart

In the left-hand column of the source code below, is a list of the objects/classes that each line of source code is translated. The right-hand column has the source code listed as it would be seen in the source code file.

Objects	Source Code
LinkList	List: QueueList
NodeNode	:Node QueueNode Head :Right
NodeNode	:Node QueueNode Tail :Ignore
LinkNode	:Insert Push
LinkNode	:Remove Pop
	End:
LinkList	Node: QueueNode
NodeNode	:Node QueueNode Next :Down
LinkNode	:Key Key
	End:
RuleDef	Rule: Push
Node	:Node QueueNode NewNode
New	:New NewNode
Set	:Set @NewNode.Key <- :NewKey
Set	:Set @NewNode.Next <- :Null
If/IfExp	:If Head != :Null
Set	:Set @Tail.Next <- NewNode
Set	:Set Tail <- NewNode
If/ElseExp	:Else
Set	:Set Head <- NewNode
Set	:Set Tail <- NewNode
	:EndIf
	End:
RuleDef	Rule: Pop
Node	:Node QueueNode NextNode
If/IfExp	:If Head != :Null
Set	:Set NextNode <- @Head.Next
Destroy	:Destroy Head
Set	:Set Head <- NextNode
If/ElseExp	:Else
Msg	:Msg "No Nodes in List"
	:EndIf
	End:

Below is the internal structure that is formed by the source code above. In the left-hand tree, the ListDef has two branches, the first branch is a list of nodes (Node Names with relative positioning) and the second is a list of rules (Rule Names). In the right-hand tree, the NodeDef contains a list of nodes and keys. The nodes contain a names of NodeDef types and identifiers, while the Keys contain an identifier for the key. **Note: NodeNode is a name of a class and not a misspelling.

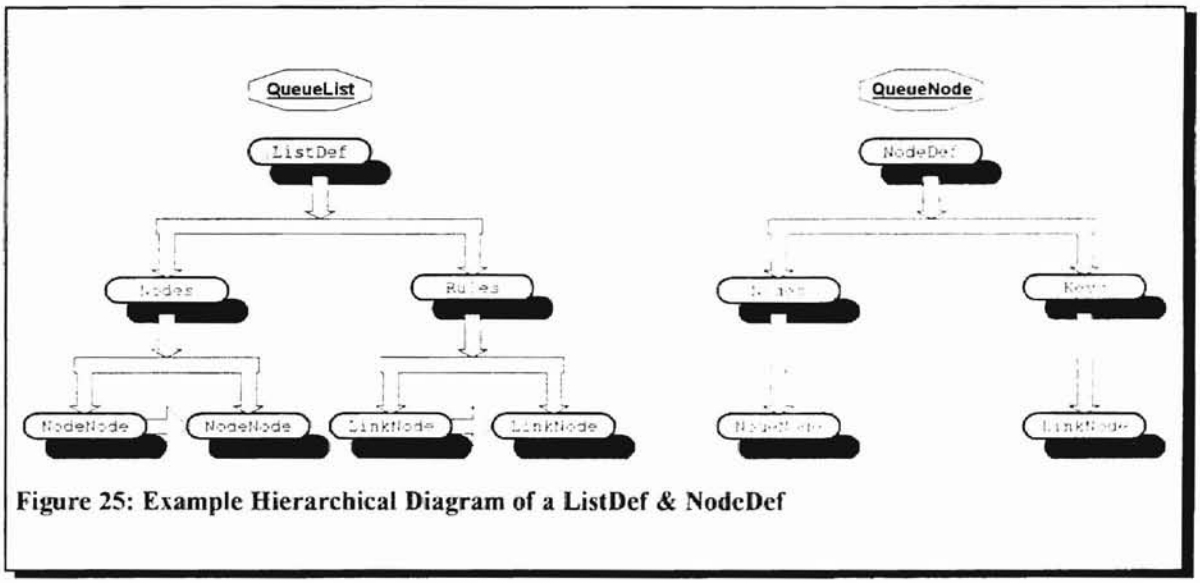


Figure 25: Example Hierarchical Diagram of a ListDef & NodeDef

The Chart below shows the relationship of the expressions in the Push rule for the example code above. The Rule, Push, contains the following objects in order: Node, New, Set, Set, and If. Inside the object of type If are two branches, IfExp and ElseExp. Both the IfExp and the ElseExp are objects of type LinkList. Each LinkList has two expressions (objects) within it. Each object inside of both the IfExp and the ElseExp is of type :Set.

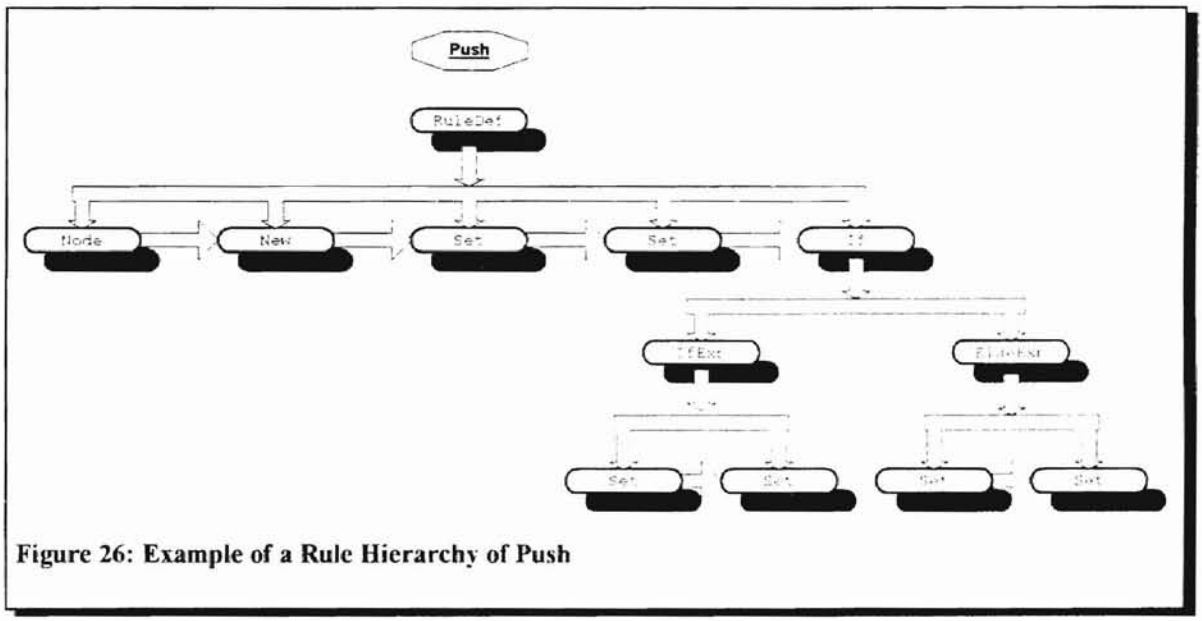


Figure 26: Example of a Rule Hierarchy of Push

As you can see by comparing the charts above and below, every method has the potential of having a unique object relationship. This relationship is in direct correlation to the source code for the rule.

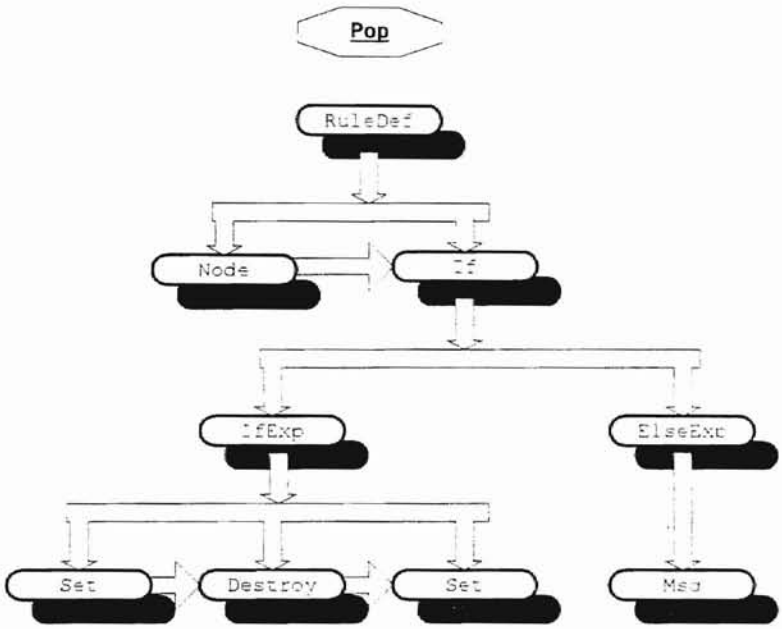


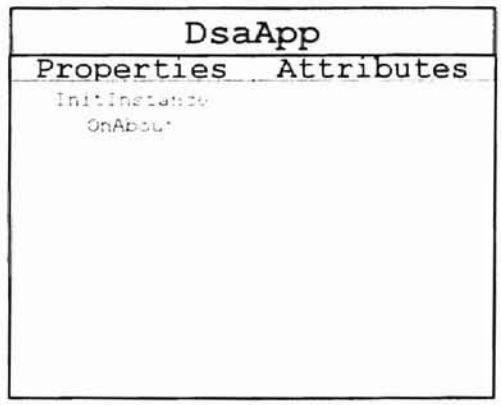
Figure 27: Example of a Rule Hierarchy of Pop

6.4 Class Descriptions

6.4.1 Class DsaApp

<u>Purpose:</u>

DsaApp is the initial class. It is DsaApp responsibility to start the entire process.



<u>Inheritance:</u> DsaApp inherits the MFC class CWinApp.

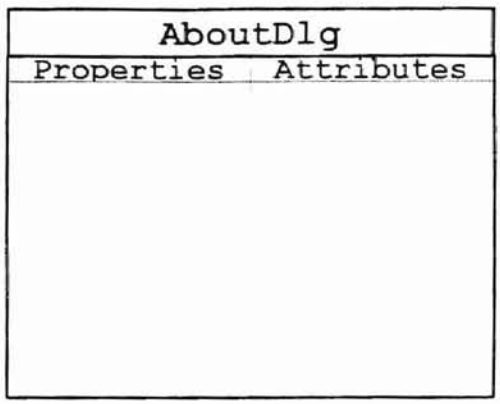
Description:

DsaApp instantiates an object of an object of type MainFrame and calls a method to display it. DsaApp also interfaces with the AboutDlg.

6.4.2 Class AboutDlg

Purpose:

AboutDlg is responsible for displaying the about box.



Inheritance: AboutDlg Inherits the MFC Class CDialog

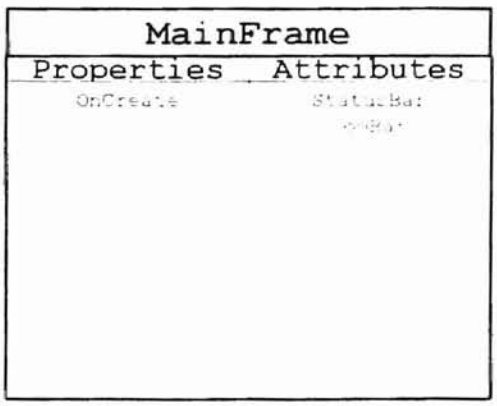
Description:

AboutDlg Displays and exits the about box.

6.4.3 Class MainFrame

Purpose:

MainFrame creates the toolbar and the status bar.



Inheritance: Inherits the MFC Class CMDIFrameWnd.

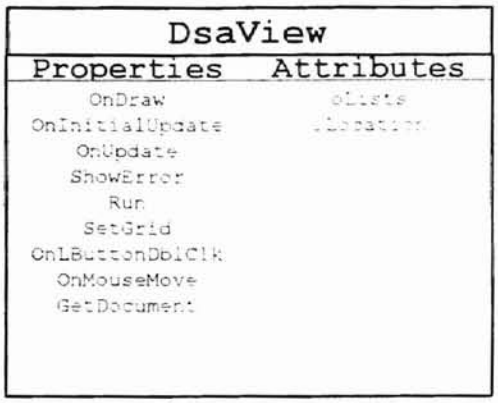
Description:

MainFrame provides the frame work for all other windows including the status bar, toolbar, and all child windows.

6.4.4 Class DsaView

Purpose:

DsaView provides all user interfaces for child windows.



Inheritance: DsaView inherits the MFC CScrollView to create child windows with actions on the child window, scrollbars.

Description:

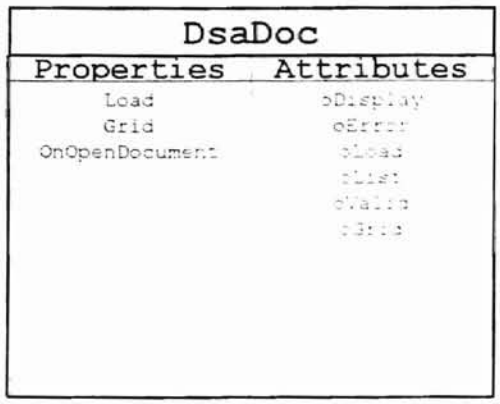
DsaView intercepts mouse commands and determines whether the user clicked on an empty region or one that is occupied. If the region is unoccupied, begin the process of creating a list. If the region is occupied by a list node, then DsaView examines the list for its list of rules and queries the user for which rule he/she wishes to run. If the region is occupied by a node then DsaView takes no action. Once either a list is choosen or a rule is run, DsaView invalidates the child window, causing the Draw method to be

called from within DsaView. From the Draw method, DsaView processes each list the user has previously created and calls its Draw method.

6.4.5 Class DsaDoc

Purpose:

DsaDoc purpose is to provide access to the Dsa program source file and the Load and Grid objects.



Inheritance: DsaDoc inherits the MFC class CDocument.

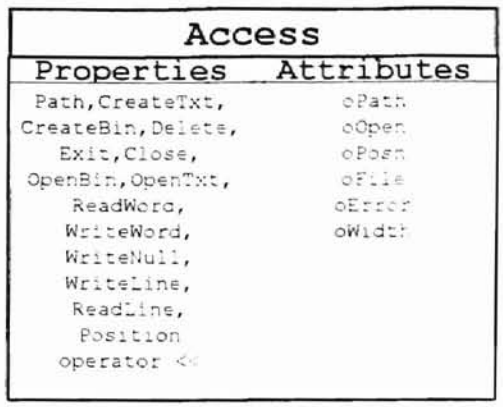
Description:

When a child window is created, the method Open OnOpenDocument is called. From inside of this method, load and grid objects are created. The load object is passed the path of the Dsa program source that OnOpenDocument received from the system. After load object completes its tasks, DsaDoc queries the load object to determine if it was successful. If not successful, DsaDoc displays an error message and terminates the child window. If the load object is successful, DsaDoc returns control back to the system. DsaDoc also provides access to the load and grid objects for use by other classes.

6.4.6 Class Access

Purpose:

Access provides easy access to data files.



Inheritance: None.

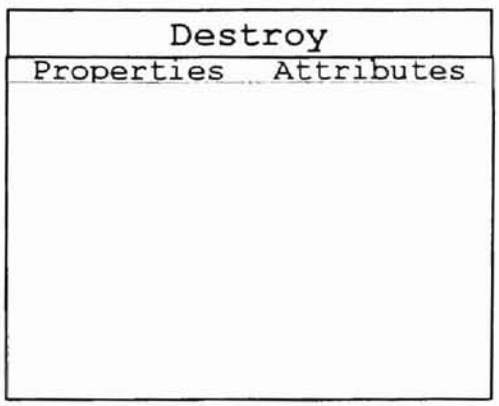
Description:

Access provides methods to open and close files, and to read and write to files in either binary or text format. Access also provides stream operations for easier use.

6.4.7 Class Destroy

Purpose:

Destroy is an Expression object used by rules. Destroy releases a node and sets a nodes properties to null.



Inheritance: RuleBase which inherits KeyWords which Inherits LinkNode.

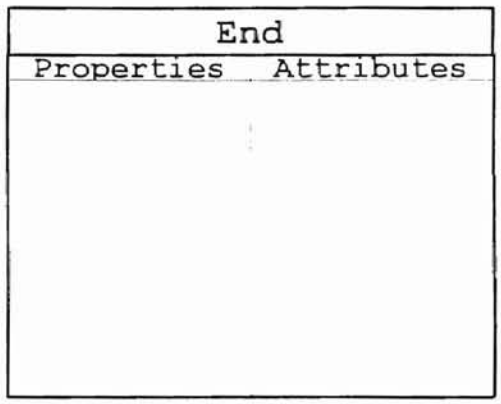
Description:

Destroy is a child of RuleBase, KeyWords, and LinkNode. Destroy uses the attributes of LinkNode to store the name of the node to be destroyed. Destroy provides the name of the node to be destroyed to the RuleBase protected method, which in turn, either destroys the named node (if direct access) or the node pointed to by the named node (if indirect access) or the referenced node if a reference.

6.4.8 Class End

Purpose:

End provides a maker that tells the RuleBase to end execution of a Rule.



Inheritance: RuleBase who Inherits KeyWords who Inherits LinkNode.

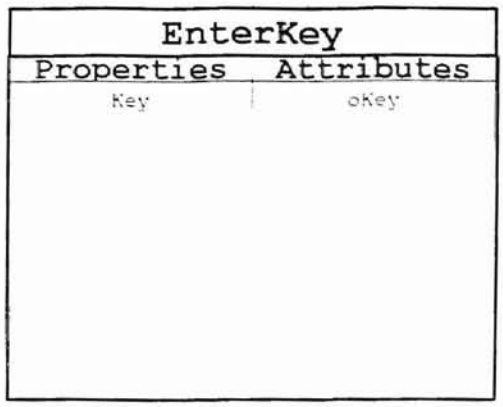
Description:

End provides a maker for RuleBase to inform it to stop execution of a Rule and return control to the system.

6.4.9 Class EnterKey

Purpose:

EnterKey queries the user to enter a key.



Inheritance: CDialog.

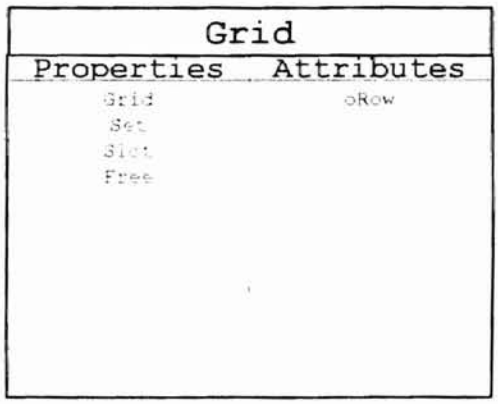
Description:

Enter Key pops up a dialog that queries the user for a key. Once entered, EnterKey provides access to the entered key data.

6.4.10 Class Grid

Purpose:

Grid provides a matrix of 25 x 25 regions. Each region can be occupied by only one object (node or list) at a time.



Inheritance: none.

Description:

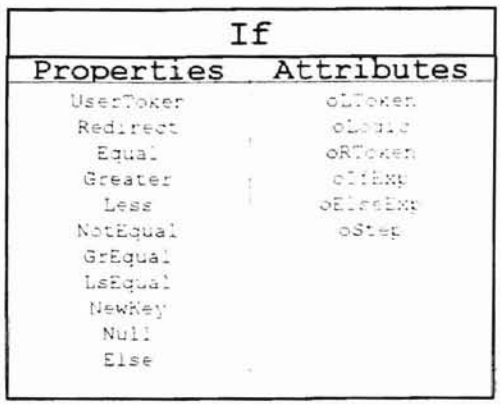
Grid provides access to an internal grid by providing methods to access the grid to either set a region to an

object, free a region from an object, or query the grid for the nearest open region to given coordinates.

6.4.11 Class If

Purpose:

If controls the branching of execution of a rule by storing and later examining the contents of a conditional.



Inheritance: RuleBase who inherits KeyWords who Inherits LinkNode.

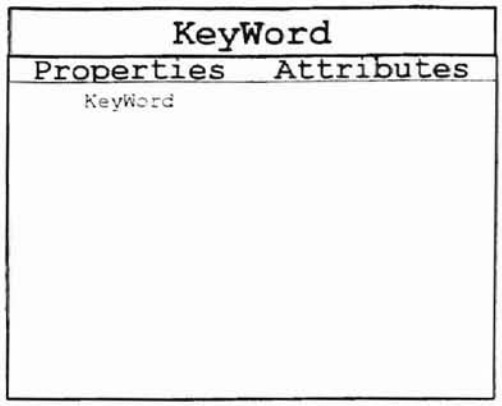
Description:

When RuleBase encounters an If object, it calls the If object run method to continue execution of a method. The run method in the If object the examines the conditional by retrieving any nodes that the condition uses, examining their contents and determining if the conditional is true or false. The If object stores two Expression lists (IfExp and ElseExp) both of class type linklist. If the condition is true, the If objects calls the inherited RuleBase method Run, passing it the first element of the IfExp list. If false, the If object passes the first element of the ElseExp to the RuleBase method Run. See RuleBase for further details.

6.4.12 Class KeyWord

Purpose:

KeyWord is an abstract base class and provides a method to match strings to keywords.



Inheritance: LinkNode.

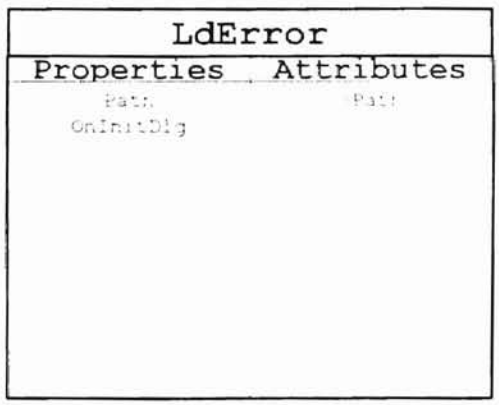
Description:

KeyWord has only one method, KeyWord. KeyWord receives a string and attempts to match it to a KeyWord. If no match is found, the string is assumed to be a user identifier.

6.4.13 Class LdError

Purpose:

Display an error dialog whenever the Load Object fails to load a Dsa Program source due to source program errors.



Inheritance: CDialog.

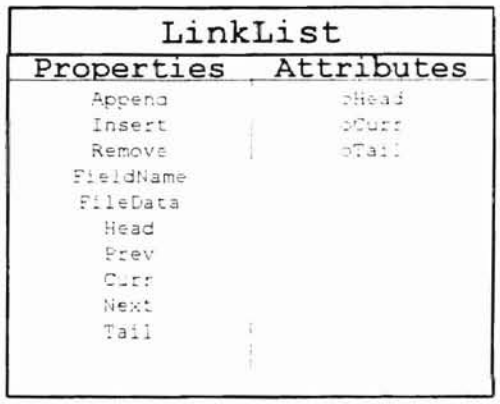
Description:

DsaDoc calls the load object to load a Dsa source program. After control is returned to DsaDoc, DsaDoc examines the Load Object to determine whether it was successful or failed. If it failed, DsaDoc passes the name of the original Dsa Program Source file to LdError and calls LdError inherited method to display the load error message.

6.4.14 Class LinkList

Purpose:

LinkList is a general purpose class that provide the properties of a link list, queue, and a stack.



Inheritance: None.

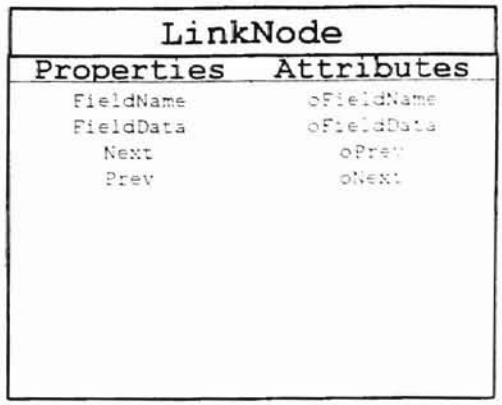
Description:

LinkList contains three pointers of type LinkNode. LinkList methods either provides access to the LinkNode pointers with the methods Head, Curr, or Tail, or LinkNode provides access to the current LinkNode (pointed to by oCurr), which includes access to the FieldName, FieldData, the Next Pointer in the LinkNode, or the Previous Pointer in the LinkNode.

6.4.15 Class LinkNode

Purpose:

LinkNode is a general purpose class that provide the properties of a node in a linked list, queue, or a stack.



Inheritance: None.

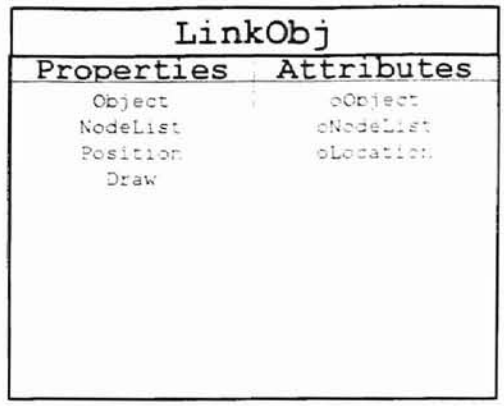
Description:

LinkNode contains two strings that hold the name of the node and the data string. In reality, these two strings could hold any string information. LinkNode also has two pointers (Prev,Next) to objects of the same class (LinkNode). All methods are used to access these four attributes.

6.4.16 Class LinkObj

Purpose:

LinkObj is used to wrap a node object. Its purpose is to provide a tree structure to draw the nodes.



Inheritance: LinkNode.

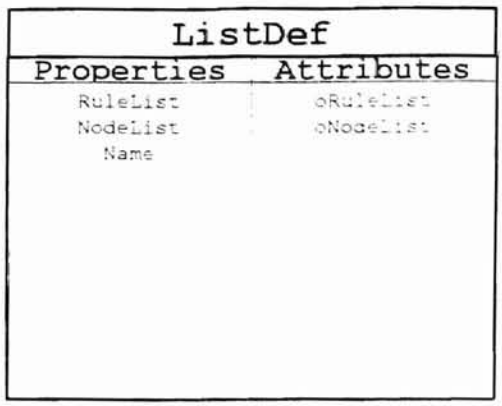
Description:

LinkObj contains pointers to an object of class type Object and a pointer to a NodeList object. LinkObj requires that an object of type Object (actually NodeObj) be used in creating a LinkObj. LinkObj then uses the Object to populate the NodeList with all children of the Object that have not already be wrapped by a LinkObj. Inside of Object is attribute called stage; every time a LinkObj wraps an Object, its stage is changed to reflect this fact. If an Object stage is already set to indicate that it is already wrapped, then LinkObj will not re-wrap it. This prevents the possibility of never-ending looping between nodes that are linked in a circular fashion.

6.4.17 Class ListDef

Purpose:

LinkDef provides the definition of a List.



Inheritance: KeyWords which Inherits LinkNode.

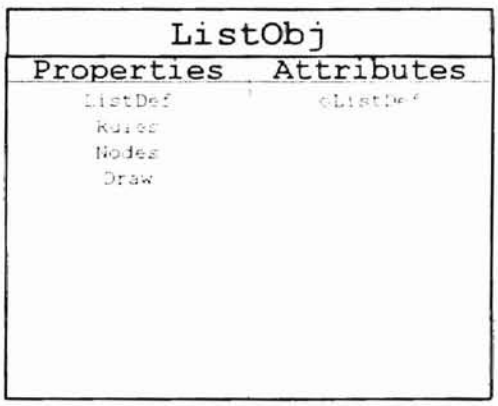
Description:

LinkDef contains the name of the List (type), and two LinkLists that contain a list of rule names and a list of node names.

6.4.18 Class ListObj

Purpose:

ListObj holds the instantiated information for a List created for the user.



Inheritance: Object.

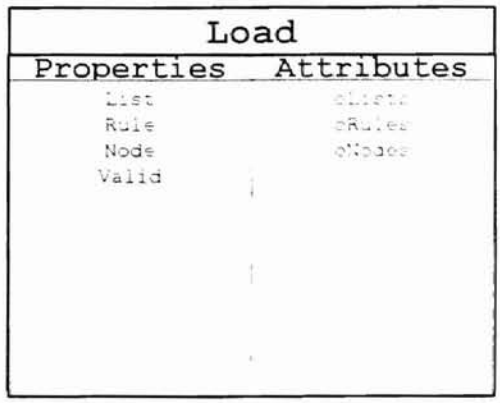
Description:

ListObj utilizes the inherited class Object to store a list of pointers to all nodes that are available to a user List. ListObj also holds a pointer to the definition of the list (ListDef).

6.4.19 Class Load

Purpose:

Load loads a Dsa source program and creates a series of ListDefs, NodeDefs, and RuleDefs.



Inheritance: KeyWords who Inherits LinkNode.

Description:

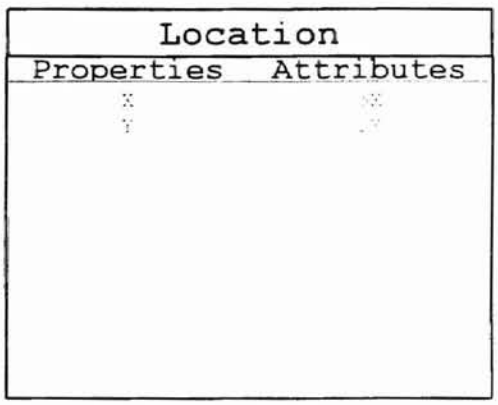
Load first creates an object of type Scan. It uses Scan to parse out tokens. It then uses the inherited class KeyWords to determine what the token represents. If the token represents either a Rule:, List:, or Node:, then load creates an object of RuleDef, ListDef, or NodeDef. Load then calls the Run method in each of these objects and passes it the object Scan. Each of these objects will in turn use scan to build up the objects until they reach a End: token. Once an End: token is reach, control is again

passed to the load object and Load continues to process the input file. Each time load creates an object, it appends it to one of three LinkLists by type. Once Load has processed all of the input file, Load will have three LinkList's each divided by object type: one list of RuleDefs, one list of ListDefs, and one list of NodeDefs.

6.4.20 Class Location

Purpose:

Location provides a two-element object that contain coordinates x and y.



Inheritance: None.

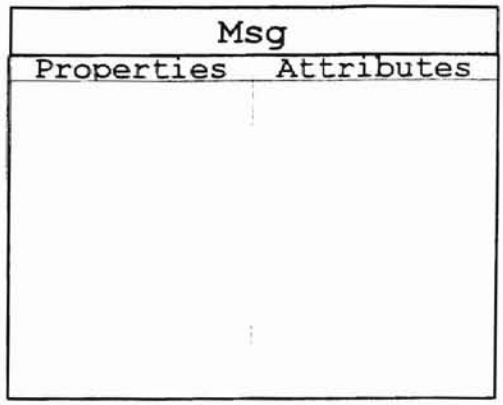
Description:

Store and retrieve X and Y coordinates.

6.4.21 Class Msg

Purpose:

Msg provides a RuleBase object for displaying a message.



Inheritance: RuleBase which inherits KeyWords which inherits LinkNode.

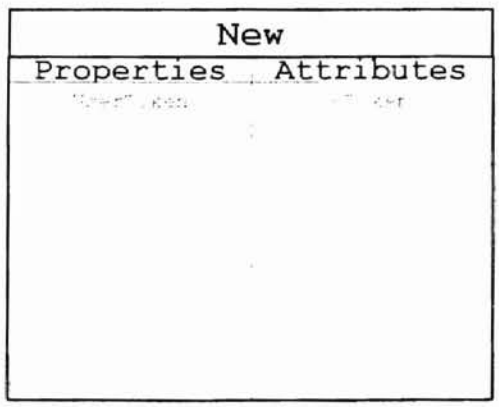
Description:

Msg is a marker that holds a string to be displayed. Once the RuleBase encounters an object of type Msg, it queries the inherited LinkNode Data pointer to provide the contents of the message. RuleBase then displays the message to the user.

6.4.22 Class New

Purpose:

New provides a RuleBase object that causes a node to receive memory.



Inheritance: RuleBase which

inherits KeyWords which Inherits LinkNode.

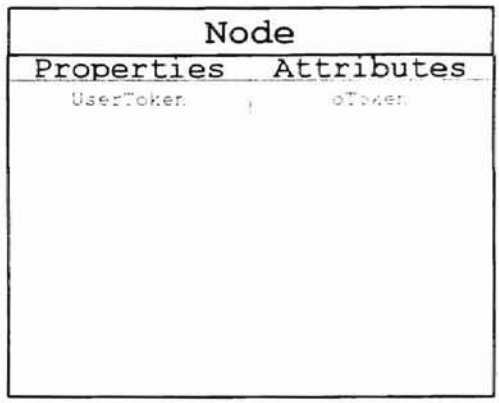
Description:

New is a marker that holds the name of the node to be given memory. Once RuleBase encounters an object of type New, it queries the inherited LinkNode Data pointer to provide the name of the node to receive memory. RuleBase then retrieves the NodeObj that the New object referred to and calls its New Method.

6.4.23 Class Node

Purpose:

Node provides a RuleBase object that causes a Rule to create a temporary NodeObj (local).



Inheritance: RuleBase which inherits KeyWords which Inherits LinkNode.

Description:

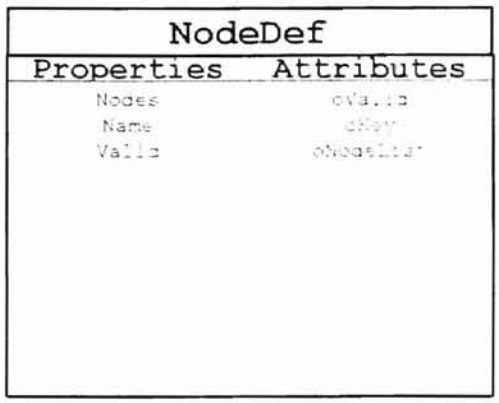
Node is a Maker that stores the name of the node type in the inherited LinkNode class. Node also stores the temporary name of the local Node object. RuleBase uses the node type

to create a temporary (local) NodeObj and gives it the name that is stored internally in the Node (oToken).

6.4.24 Class NodeDef

Purpose:

NodeDef provides the definition of a Node.



Inheritance: KeyWords which Inherits LinkNode.

Description:

NodeDef contains the name of the node types and two LinkLists that contain names of node types and key names.

6.4.25 Class NodeList

Purpose:

Provides a list of nodes to be displayed, used only during the drawing.

NodeList	
Properties	**Attributes**
Append Location Position Draw	oUp, oNumUp oUpRight, oNumUpRight oRight,oNumRight oDownRight oNumDownRight oDown,oNumDown oDownLeft oNumDownLeft oLeft,oNumLeft oUpLeft,oNumUpLeft

Inheritance: LinkList.

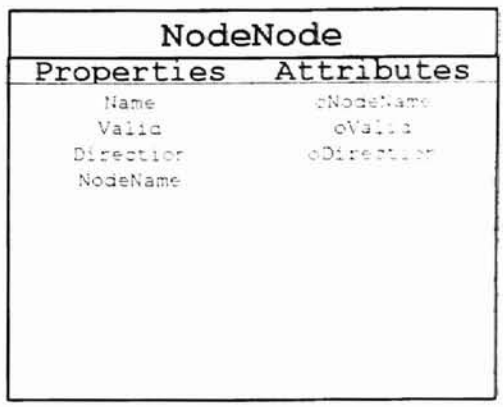
Description:

Every time DSA draws, it builds a set of nodes using NodeList and LinkObj. NodeList builds a hierarchy of node pointers where there can only be one node pointer per hierarchy. Each time a NodeObj is added to a node list, its status is changed from stage 0 to stage 1. When the NodeObj is printed, it is changed back to stage 0. Any node that is not at stage 0 is not added to the NodeList.

6.4.26 Class NodeNode

Purpose:

NodeNode is used to store the name of a node inside of a ListDef, NodeDef, or a temporary node inside of a rule. It is different from a NodeDef in that it does not hold the definition but holds the node type name, name of the node pointer, and the relative direction.



Inheritance: KeyWords which Inherits LinkNode.

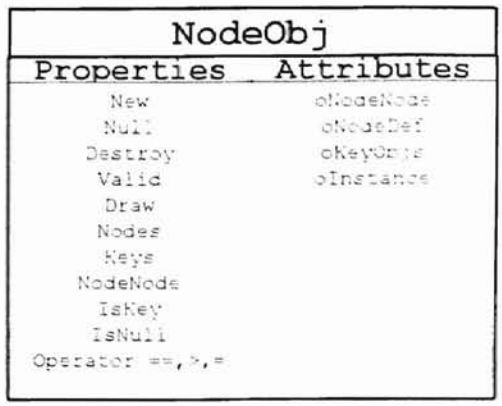
Description:

NodeNode stores three values, the name of a node type, the name of the node pointer, and the relative direction (relative direction is not used for temporary nodes in a rule). During run time, the name of the node type must match the name of a node type in a NodeDef or a run time error will occur.

6.4.27 Class NodeObj

Purpose:

The NodeObj holds the instantiated information for a Node created by a rule. The node object may be a temporary node inside of a rule, a node, a node inside of a list.



Inheritance: Object.

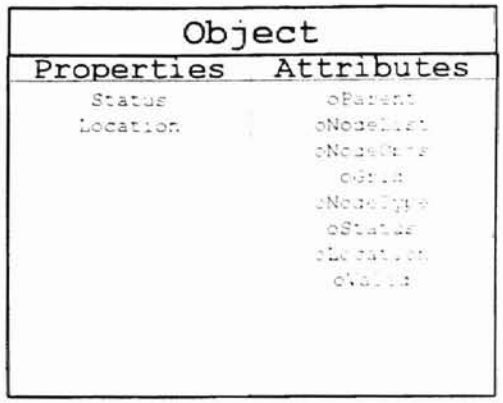
Description:

NodeObj utilizes the inherited class Object to store a list of pointers to all nodes that are available along with all the key objects. A NodeObj can have one of three different node types: Pointer, Key, or Reference. If the Node Object is a pointer, then it points to a list of instantiated nodes and keys. If the NodeObj is a reference, then it points to another NodeObj and utilizes its internal lists.

6.4.28 Class Object

Purpose:

Object serves as the base class for ListObj and NodeObj. Its purpose is to store similar information.



Inheritance: KeyWords which Inherits LinkNode.

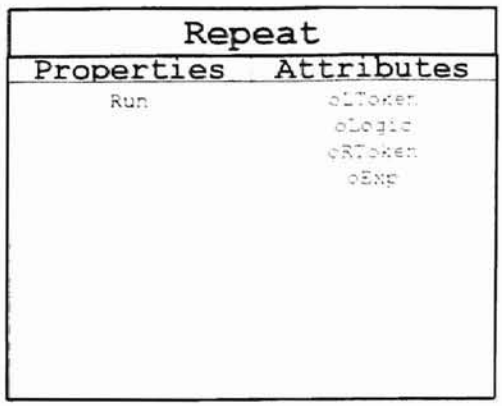
Description:

In both NodeObj and ListObj are stored a list of nodes. The list of nodes are defined in the mutual parent class of Object. Object stores a NodeType that indicates if the object is a list, a node, a key, or a reference to a node.

6.4.29 Class Repeat

Purpose:

Repeat controls the branching of execution of a rule by storing and later examining the contents of a conditional.



Inheritance: RuleBase which inherits KeyWords which Inherits LinkNode.

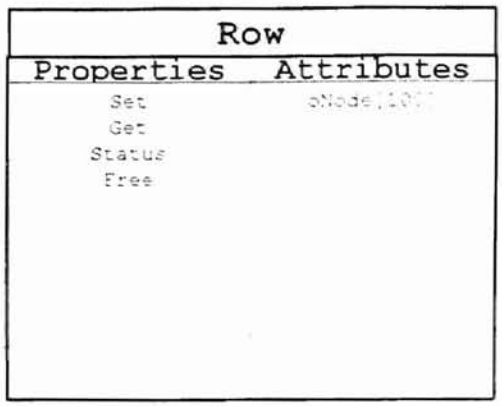
Description:

When RuleBase encounters a Repeat object, it calls the Repeat object run method to continue execution of a method. The run method in the Repeat object examines the conditional by retrieving any nodes that the condition uses, examining their contents and determining if the conditional is true or false. The Repeat object stores an Expression lists (oExp) of class type linklist. If the condition is true, the Repeat objects calls the inherited RuleBase method run, passing it the first element of the oExp list. After the run method has completed and returned control back to the Repeat object, the Repeat object loops back and rechecks the conditional. If the condition is false, the Repeat object returns control to the calling object. See RuleBase for further details.

6.4.30 Class Row

Purpose:

Row provides a matrix of 1 x 25 regions. Each region can be occupied by only one object (node or list) at a time. 25 Rows make a Grid.



Inheritance: none.

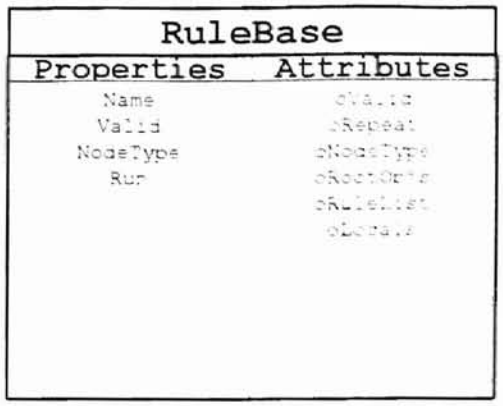
Description:

Row provides access to an internal row of a grid by providing methods to access the row to either set a region to an object, free a region from an object, or query to determine whether a region is free and available.

6.4.31 Class RuleBase

Purpose:

RuleBase is the base class for all expressions and rules.



Inheritance KeyWords who Inherits LinkNode.

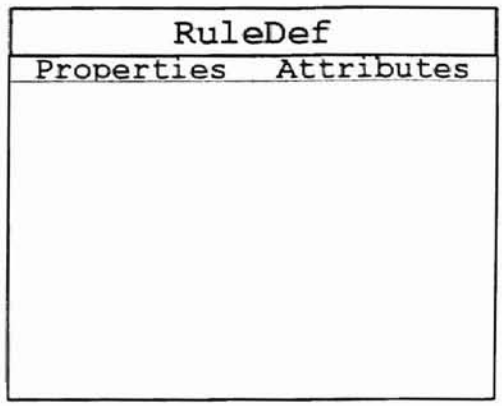
Description:

RuleBase provides a virtual method for every keyword. If a keyword is a valid operation in all rules, it performs that method. However, for most operations, RuleBase provides an error message. It is the responsibility of the inherited Expressions to override any operation that it considers valid. In addition to providing virtual methods for each keyword, RuleBase provides a run method that loops through a list of nodes, checks their type and calls the appropriate method. In this way, RuleBase contains the only methods for moving from one expression to another.

6.4.32 Class RuleDef

Purpose:

RuleDef provides the definition of a Rule.



Inheritance: RuleBase which inherits KeyWords which Inherits LinkNode.

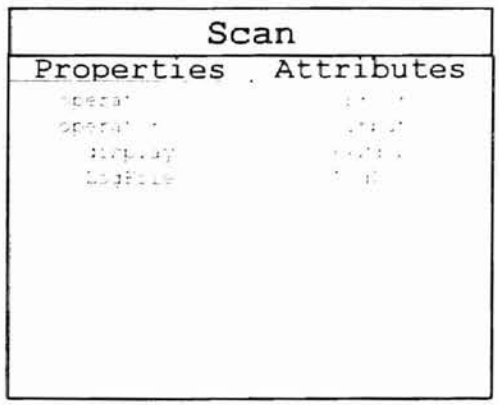
Description:

RuleDef contains the name of the rule and a LinkList that contains the expressions for a rule. RuleDef utilizes the base class RuleBase to execute the expressions in a rule.

6.4.33 Class Scan

Purpose:

Scan processes the input file by breaking the input data into tokens. Scan also writes out two temporary files: Log File and Tmp(debug) file.



Inheritance: Display.

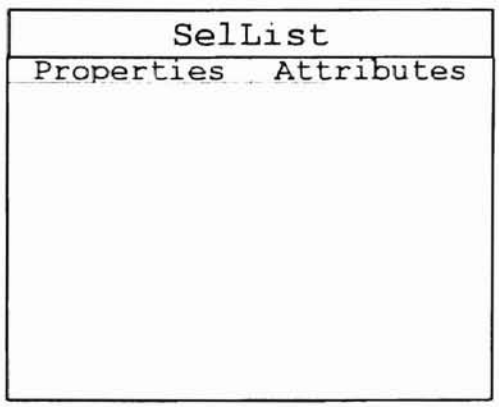
Description:

Scan opens up the input file and creates a log file and a tmp file. Each time scan encounters an end-of-line or end of file, scan writes the line out to the tmp file along with a line number. Scan opens the log file but does not use it, but rather passes it to the load object which passes it on up.

6.4.34 Class SelList

Purpose:

SelList Displays all the lists types that are available from the DSA program source file.



Inheritance: CDialog.

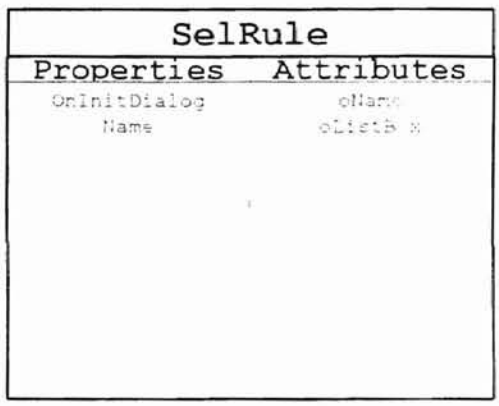
Description:

When the user double-clicks on a free region on the child window, DsaView creates the SelList object for the SelList class and passes it the names of the the ListDefs. The user then must select a list from the a listbox and either clicks ok or double-click on the name. This name is returned to DsaView where a list is displayed on the child window.

6.4.35 Class SelRule

Purpose:

SelRule Displays all the rules that are available for a particular list.



Inheritance: CDialog.

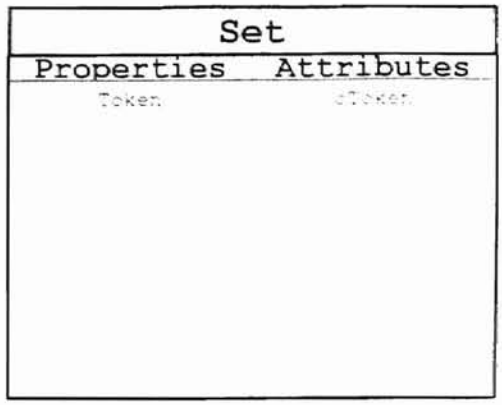
Description:

When the user double clicks on a region on the child window that contains a list, DsaView retrieves the list of rules from the ListDef and displays them, then passes them to SelRule. Then DsaView calls a method in SelRule to display the list of rules. When the user selects a rule, control is passed back to DsaView along with the name of the rule, and DsaView processes the rule.

6.4.36 Class Set

Purpose:

Set provides an expression to set the value of the value of a node.



Inheritance: RuleBase which inherits KeyWords which Inherits LinkNode.

Description:

Set stores the name of the node to be set, the operation to be performed, and the value used to set the node (or the name of the node to be used).

7. Future Direction

While Dynamic Data Structure Animation is a very interesting and rewarding subject, it appears that it is generating very little interest in the computer science community. More emphasis is being placed on static representations than dynamic, and almost exclusively in association with compilers. Algorithm animators also are generating interest, but because of the large scope, it is more difficult to produce data structure animation.

The language that was created for DSA should be expanded to include such important language features as methods (to be called from rules), so that recursion can occur, as well as integer data types, strings, arrays, and other structures besides nodes and lists. While the current language allows for simple structures (and some not-so-simple structures) such as linked lists, binary trees, three trees, stacks, queues, etc.., it would have a difficult time animating AVL trees, Red-Black trees, or any other tree that requires a rank. DSA would also have a difficult (if not impossible) time animating hashes and heaps that were not trees. By adding these features in the futures, such structures would no longer be difficult or impossible to generate.

Another feature that would be helpful for DSA would be the ability to convert DSA code to 'C','C++', and JAVA code. In this way, the user could build his or her structure relationships in DSA and later use them in subsequent programming efforts.

It would also be useful to have an interactive debugger and the ability to write and edit DSA code from within the application, instead of having to use a separate text editor. It would also be useful if a trace was built into DSA so that the user could step through the DSA code. While these would all be very helpful features, they would also be very time-consuming to implement.

8. Conclusion

"A picture is worth a thousand words."

Author unknown.

DSA gives a visual representation to an abstract concept. For many users exploring data structures, whether they are students, researchers, or program developers, DSA gives a concrete physical image to abstract concepts. For students learning a data structure, the ability to see how their structure interacts is a very important learning tool. For the researcher designing new data structures, the ability to explore visually the data structure is invaluable. For the developer, the ability to visually observe his/her data structure could potentially save many hours of development time.

DSA is an important tool that improves on data structure animation that is currently not available. While there are many algorithm animators that are capable of animating data structures, their focus is so broad that they can be cumbersome to use to create a data structure animation. Because DSA focus is exclusively on Data

Structures, it is able to narrow its focus and therefore make it simpler to animate data structures. For animating a dynamic data structure, I believe a DSA is the best tool for the job.

Selected Bibliography

[ARR92] Arra, Shravan K., "Object-Oriented Data Structure Animation," Graduate College of Oklahoma State University, Master Thesis, Computer Science, July 1992.

[BAE74] Baeker, R.M., "Genesys Ineractive Computer-Mediated Animation", In Computer Animation, J. Halas, Ed., Hastings House, New York, N.Y. pp 97-115, 1974

[BAL90] Balci, Osman, Nance Richard E., Derrick, E. Joseph., Page, Ernest H., Bishop, John L., "Model Generation Issues in a Simulation Support Environment", Technical Report TR-90-40, Department of Computer Science and System Research Center, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, pp. 13, 1 August 1990

[BAE81] Baeker, Ronanld M., "Sorting out Sorting", 16mm color sound film. 1981

[BOO75] Booth, K., "PQ-Trees", 16mm color silent film, 1975

[BRO87] Brown, Marc H., "Algorithm Animation," An ACM Distinguished Dissertation 1987, The MIT Press, 1988.

[BRO92] Brown, Marc H., "Zeus: A System for Algorithm Animation and Multi-view Editing," System Research Center, Digital Equipment Corporation, February 28, 1992.

[BRO93a] Brown, Marc H., "The 1992 SRC Algorithm Animation Festival," System Research Center, Digital Equipment Corporation, March 27, 1993

[BRO93b] Brown, Marc H., and Nojork, Marc A., "Algorithm Animation Using 3D Interactive Graphics," SRC Research Report, Digital Equipment Corporation, September 15, 1993

[CHA86] Edited by: Chang, Shi-Kuo, Ichikawa, Tadao, and Ligomendies, Panos A., "Visual Languages," Management and Information Systems, Plenum Press, New York and London, 1986

[CON79] Conway, Richard., and Gries, David , "A Structured Approach Using PL/I and PL/C, Third Edition," Winthrop Publishing, Inc. 1979.

[COP92] Coplien, James O., "Advanced C++, Programming Styles and Idioms", AT&T Bell Laboratories, Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[FEL88] Feldman, Micheal B., "Data Structures with Modula-2", George Washington University, Prentice Hall, Englewood Cliffs, N.J. 1988

[HIL88] Hille, Reinhold F., "Data Abstraction and Program Development Using Pascal", Department of Computer Science, The University of Wollongong, Australia, Prentice Hall, 1988.

[KNO66] Knowlton, Kenneth C., "L6: Bell Telephone Laboratories low-level linked list language", two black and white sound films, 1966.

[KNU73] Knuth, Donald E., "The Art of Computer Programming, Second Edition," Addison-Wesley Publishing Company, 1973.

[LAW94] Lawrence, Andrea W., Badre, Albert, and Stasko, John T., "Empirically Evaluating the Use of Animations to Teach Algorithms," Technical Report GIT-GVU-94-07, Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Altanta, GA 30332-0280, July 1994.

[MAK91] Mak, Ronald, "Writing Compilers and Interpreters", John Wiley & Sons Inc., 1991.

[MAN89] Manber, Udi., "Introduction to algorithms, A Creative Approach " University of Arizona, Addison-Wesley Publishing , 1989

[MAR86] Marcus, Caudia., "Prolog Programming,", Addison-Wesley Publishing Company, Inc., 1986

[MUF82] Mufti, Aftab, A. "Elementary Computer Graphics", Reston Publishing Company, Inc., 1982.

[RAL2nd] Ralston, Anthony., Reilly, Edwin D., Jr., "Encyclopedia of Computer Science and Engineering, 2nd Edition," Van Nostrand Reinhold Company, New York, N Y.

[STA92] Stasko, John T., and Turner Carlton R. "Tidy Animations of Tree Algorithms," Technical Report GIT-GVU-92-11, Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Altanta, GA 30332-0280, November 1992

[THA90] Edited by: Thalmann, Daniel, "Scientific Visualization and Graphics Simulation," John Wiley & Sons, Inc., New York, 1990.

[TAL95] Tal, Ayellet and Dobkin, David. "Visualization of Geometric Algorithms," IEEE Transactions on Visualization and Computer Graphics, Vol. 1, No. 2, Page 194-204, June 1995.

[VIN84] Vince, John., "Dictionary of Computer Graphics", Knowledge Industry Publications, Inc., White Plains, NY 1984

[WEI93] Weiss, Mark A., "Data Structures and Algorithm Analysis in C," The Benjamin/Cummings Publishing Company, Inc., 1993.

[YOU89] Yourdon, Edward, "Modern Structured Analysis," Yourdon Press, Englewood Cliffs, New Jersey, 1989.

VITA

Lee Hou Harvick

Candidate for the Degree of

Master of Science

Thesis: RULE BASED DATA STRUCTURE ANIMATION

Major Field: Computer Science

Biographical:

Education: Received Bachelor of Science degree in Mechanical Engineering from Tongji University, Shanghai, China, in July 1987. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May, 1997.