Tool for Structural Testing of Rule-based System

Piyapattana Temchareon
Miami University, commons-admin@lib.muohio.edu

TECHNICAL REPORT: MU-SEAS-CSA-1993-009

Tool for Structural Testing of Rule-based System

Piyapattana Temchareon

Tool for Structural Testing of Rule-based System

by

Piyapattana Temchareon
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #93-009 May, 1993

Tool for Structural Testing of Rule-based System

by

Piyapattana Temchareon

Department of System Analysis
Miami University
Oxford, Ohio 45056

January, 1993

TABLE OF CONTENTS

Tool for Structural Testing of Rule-based Expert System

PIYAPATTANA TEMCHAREON
Miami University

Traditional software testing techniques are useful in testing some components of the expert systems, such as the inference engine. However, there is a need for the development of a testing method that tests the structure in a rule-based expert system. This testing method should be computationally effective, provide a reasonable level of coverage of the rule base, and be independent of the inference engine [KIPER92]. In the article *Structural testing of Rule-based Expert Systems* by Dr. James D. Kiper, the logical path graph (LPG) method was chosen as a practical testing method for an expert system. However, the logical path graph algorithm developed in that paper requires further investigation by application to large rule bases. This need has resulted in the development of a computer tool called Graptool. The Graptool software, that has been created, was based on the logical path algorithm and was written in the C language. This tool reads the CLIPS rule-based expert system as a text file, and constructs a listing that represents the structure of the rule base.

Key words and phrases: See appendix A.

1. INTRODUCTION

Expert system (ES) is a branch of artificial intelligence that uses specialized knowledge extensively to solve problems at the level of the human expert. It is a computer system that mimics the decision-making ability of these human experts. Expert systems have a number of attractive features that give them many advantages over other systems. First of all, expert systems (like a mass production of experts) increase the availability of expertise on any suitable computer hardware. Secondly, there is reduced cost (for expertise used) and reduced danger for the human expert (ES can be used in environments that might be hazardous for humans). The expertise in an expert system is permanent in that its knowledge will last indefinitely, and the areas of expertise can be multiple. The knowledge of several experts can be made available to work simultaneously and continuously on a problem at any time. By providing second opinions and breaking ties between two human experts, an ES provides increased reliability by assuring confidence that a correct decision was made. Furthermore, expert systems may respond faster, and be more available than the human expert. They can act as an intelligent tutor and can be used to access a data base in an intelligent way. Since the knowledge in expert systems is explicitly specified, it can be examined for correctness, consistency and completeness, whereby it may be modified. This may, in turn, improve the quality of the decision making [RIL89].

Research Objective

The testing of a rule base is a technique that tests the structure of each new rule base developed for use with a specific inference engine. This type of structural testing is important to help assure the correct operation of the expert system. Even though the inference engine testing is effective and reliable, the distribution of knowledge of an ES into a collection of rules requires the development of new testing techniques for this rule base. The testing method of this research project graphically represents a rule base in a way that is analogous to control flow graphs of traditional software. In addition, this graphic form can then characterize the complexity of its rule base and can be used to determine a set of paths through it that adequately tests the rules and their interaction. In order for this structural testing to be done successfully, this method will construct the **logical path graph** of a rule base. See figure 9 for a pictorial description of the logical path graph algorithm.

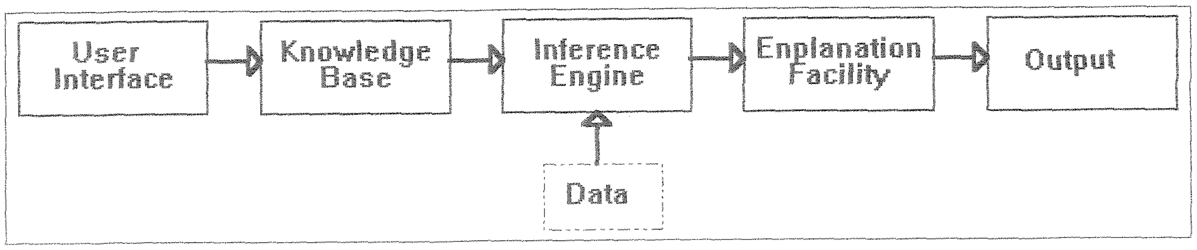
The goal of this project is to develop a computer tool based on the logical path graph method. The computer tool that is written in the C language is called **Graptool**. It will allow the logical path graph algorithm to be applied to the rule-based expert systems. The result of this experiment will determine whether the logical path graph method is useful for testing any rule base structure. Even though the logical path graph algorithm cannot provide a complete solution to the need for validation and verification of a rule-based expert system, it does provide a technique that can reduce the risk associated with its use [KIPER92].

In order to develop the software, a test bed was selected. Although this test bed could be any expert system shell that supported rule-based development, CLIPS was selected for use with the software tool because it was used in Dr. Kiper's experiment, its syntax and semantics are very close to those of some other very popular commercial systems, it was available at no cost since it is a product of NASA at the Artificial Intelligence Section of the Johnson Space Center (JSC), and its source code is available giving access to the internal structure of the inference engine, rule base, or fact base [KIPER92].

This paper contains the following: an introduction to expert systems, the CLIPS expert system shell, the logical path graph algorithm, each component of Graptool works, and the Graptool source code.

2. AN EXPERT SYSTEM OVERVIEW

Expert system technology is an advanced software tool that provides programmers with new programming environment for incorporating symbolic representation of facts, data, and heuristic knowledge in their conventional software. The major components of an expert system are the knowledge base, inference engine, user interface mechanism, and data. The structure of an expert system is shown below:



2.1. *Knowledge base*

An expert system can be built based on three types of knowledge bases: rules of thumb; facts and relations among components; and assertions and questions. The components that represent these types of knowledge bases are called rules, frames, and logic [HU89]:

a. **Rules:** Rules are used to represent rules of thumb for the knowledge base. Rules are conditional sentences that are expressed in the following form:

IF ***(premise)*** *FACT1, . . . FACTn THEN* ***(conclusion)*** *FACT1, . . . FACTn*

b. **Frames:** A frame provides a convenient structure for representing objects that are typical to a given situation. It allows nodes to have structures that can be simple values or other frames. It can be computed through procedures or other computer programs, and has been used to represent the structure and relation of the facts.

c. **Logic:** Logic is the expression of predicates and values to model facts of the real world. Logic is good for representing assertions and queries.

2.2. *Inference engine*

Before an ES can be executed using a knowledge base, a reasoning and searching mechanism must be included for controlling the knowledge. The reasoning and searching control is the inference engine. The reasoning method in the expert system is the simple logic rule such as IF . . . ELSE or nested-IF. The searching methods are used to determined where to start the reasoning process and which rule to examine next. Two main searching methods are forward chaining and backward chaining.

Forward chaining:

In forward chaining, when the premise clause matches the situation, then the conclusion clauses are asserted. The rules that have been used already are not used again in the same search. However, the facts or the results of those rules will be added to the knowledge base. The above process will be repeated until no matching rule exists.

Backward chaining:

Backward chaining proves the hypotheses from the facts. If the goal is to determine the fact in the hypothesis, then the premises will be evaluated to determine whether or not it matches the situation.

2.3. User interface

This man-machine interface mechanism produces the dialogue between the computer and the end user. It contains the explanation module that allows the end user to understand and examine the reasoning process and its answer. The user interface also contains the groups of instruction which will guide the user through the operation of the expert system.

3. CLIPS EXPERT SYSTEM TOOL

CLIPS (the C Language Integrated Production System) was designed by NASA at Johnson Space Center with the specific purpose of providing high portability, low cost, and easy integration with external systems. It is a forward chaining, rule-based language that has inferencing and representation capabilities similar to OPS5 [GIA89].

3.1. Basic Components of CLIPS

The basic components of a CLIPS ES are the rules, the facts, and the inference engine. The following information and details about these three components are necessary to describe since the software was developed based on these concepts and rules in CLIPS.

3.1.1. Rules:

The primary method of representing knowledge in CLIPS is use of a **rule** that is defined as a set of conditions (left-hand side, LHS), and actions (right-hand side, RHS) that are to be taken if the conditions are met. In order to describe how to solve a problem, an expert system developer must define these rules. A **knowledge base** (rule base) includes the entire set of rules in an expert system.

3.1.2. Facts:

The basic form of data in a CLIPS system is called a **fact**. Each fact represents a piece of information that is placed in the current list of facts called the **fact-list** (fact base). The presence or non-existence of these facts causes the rules to execute or **fire**. Facts are made up of fields that are either a **number**, **string**, or **word**. All **numbers** are defined as any field that consists only of numbers from 0-9, a decimal point, a plus and minus sign, and an e for exponential notation. Example of a numbers are 222, and +35.9. A **string** is a set of characters that starts and ends with double quotations. A string will only match another string. It will not match a **word** or **number**. Examples of a string is "123.9", and "top".

A **word** is any sequence of printable ASCII characters followed by zero or more characters such as too, top, string, and Tim. Some of these characters act as delimiters that are used to determine the end of the word. The delimiters include the following: a space, a tab, a carriage return, a linefeed, a double quotation, parentheses, an ampersand (&), a bar (|), a less than sign (<), and a tilde (~). CLIPS is case-sensitive and some printable ASCII characters cannot be used as the first character in a word depending on the characters that follow it. The exceptions and their conditions are the following:

1) The word field cannot start with a ? or $?.
2) The field that starts with or without a sign (plus or minus) followed by numbers 0-9 or decimal is considered a number field. Anything else is considered a word field.
3) A colon followed by a delimiter means that the next item is a function call. If it is followed by anything else, it is considered a word field.
4) When & and | is followed by itself, it is considered a word. If it is followed by anything else, it acts as a delimiter.
5) The < follow by - is a special symbol. The < followed by anything else is a word.

3.1.3. Inference engine:

CLIPS uses a mechanism called the **inference engine** to determine which rules match the current facts. When the inference engine finds rules whose LHS conditions match facts, it places this rule on the **agenda** of rules to be fired. It then determines which rule on the agenda to fire. The process of matching the LHS of rules to facts is called **unification**.

3.2. The CLIPS Cycle of Execution

CLIPS is ready to execute rules once a knowledge base is built and a fact list is prepared. The starting point, the stopping point, and the sequence of operations do not need to be defined explicitly because the knowledge (rules) and the data (facts) are separated. The inference engine is used to apply the knowledge to the data. The CLIPS cycle of execution follows three basic steps:

1) Examination of the knowledge base to see if any rule conditions are met.
2) Rules whose conditions are met are placed on the agenda. The agenda is a priority queue onto which rules to be fired are pushed when matched. When several rules qualify to be fired at a given point in time, it is the job of the inference engine to determine the priority among these rules (rules of higher priority are placed above ones of lower priority).
3) The top rule on the agenda is selected and its RHS actions are executed. This execution may change the fact base so that the inference engine repeats evaluation of the rule base to determine which rules may now be added to the agenda. The cycle of rule base evaluation and rule execution continues until the agenda is empty or until the rule limit is reached.

3.3. The CLIPS Defining Constructs

There are three defining constructs found in CLIPS: **deffacts**, **defrule**, and **deftemplate**. All these constructs is opened with the left parenthesis and closed with a right parenthesis. Any pattern or block within a construct is also opened and closed with a parenthesis.

3.3.1. *Deffacts construct*

Any number of initial facts can be added to the fact-list by using a **deffacts** construct. This initial fact can be deleted or matched like any other fact. By using a **reset** command in CLIPS, all initial facts will be reconstructed and inserted into the fact-list. CLIPS also automatically adds an *(initial-fact)* to the beginning of the fact-list if the *(initial-fact)* does not already exist in the fact-list. The syntax of deffacts is the following [CUL89]:

```
(deffacts <name> ["<comment>"]
          [(<fact1>)
              .
              .
          (<fact n>)])
```

where <name> needs to start with an alphabetic character and must be included in the construct. A comment is optional and must begin and end with quotation marks.

3.3.2. Defrule construct

Each rule in the CLIPS rule-based will be defined by using the **defrule** construct. The syntax of defrule is the following [CUL89]:

```
(defrule <name> ["<comment>"]
        [(<first pattern>)
                 .
                 .        ;Left Hand Side
                 .
        (<nth pattern>)]
    =>
        [(<first action>)
                 .
                 .        ;Right Hand Side
                 .
        (<mth action>)])
```

where <name> must start with an alphabetical character and must be included in the defrule construct. A comment is optional and it can be any **string**. An **arrow** (=>) is used to separate the LHS from the RHS.

Additional details about the LHS (condition) and the RHS (action) of the rule base can be found in the next two following sections. However, only the LHS and the RHS syntax that was used in the Graptool program will be mentioned in this paper. More complete information about the LHS and RHS of the CLIPS rule base can be found in the CLIPS manual [CUL89].

3.3.3. Defining fact template

To access the facts that are *free form* and encode information positionally, the end user needs to know where data is stored and which field contains the desired data. The fact template will allow the user to abstract the structure of a fact by assigning names to each field in the facts. However, the fact template is not used in the Graptool software.

3.4. The CLIPS Left Hand Side Rule Syntax -- Condition

The left-hand side (LHS) is a series of patterns that represent the conditions that must be satisfied before a rule can fire. There are several concepts that have been used for matching and manipulating facts. These concepts involve use of literal patterns, variables (single and multiple wildcards, and single and multiple variables), and field constraints. For controlling the execution of rules, CLIPS will use the logical pattern operators (constraint patterns).

3.4.1. Literal patterns

A literal pattern is a simple matching pattern that precisely defines the exact matching between the pattern on the LHS rule with the fact in the fact base. This type of pattern does not contain any variable or wildcard fields that will be explained later. For example [CUL89]:

Pattern on LHS of rule	Fact in the fact list	Matches?
(group left town)	(group left town)	Y
(group left town)	(group in town)	N

3.4.2. Wildcards single and multifield

Rather than having to specify a specific field of a fact to trigger a rule, a general pattern can be specified by using a wildcard [RIL89]. In CLIPS, there are single and *multifield* wildcards. The ? (question mark) represents a single wildcard and $? represents the multifield wildcard. The single wildcard will match any **number**, **word**, or **string** stored in exactly one field in a fact. The multifield wildcard will match any **number**, **word**, or **string** in zero or more fields in the fact. Single and multifield wildcard symbols can be used in any combination in a single pattern. For example [CUL89]:

	Pattern on LHS of rule	Fact in the fact list	Matches?
Single	(data red ?)	(data red)	N
Wildcard:	(data red ?)	(data red green)	Y
Multiple	(data red $?)	(data red)	Y
Wildcard:	(data red $?)	(data red green)	Y
Combination:	(data ? $?)	(data red green green)	Y
	(data ? $?)	(data green red)	Y

3.4.3. Variable single and multifield

A variable retains the value of a field replaced by a wildcard. The rule for pattern matching in variables is similar to that in wildcards. On its first appearance, a variable acts just like a wildcard and stores the matching value(s) as its initial value(s). A single field variable will capture a field in the fact that matches a single wildcard. A multifield variable will capture a field (zero or more) which matches with a multifield wildcard. After that, all reference to that variable must match the value of that variable(s). This applies to both single and multifield variables. The binding will only be true within a rule in which it occurs because the variable is local to the rule. The syntax of variables are the following [CUL89]:

?<name>	;a single field variable
$?<name>	; a multiple field variable

where <name> must start with an alphabetic character. A double quote cannot be used as part of a variable name.

3.4.4. Constraining fields

Field constraints are functions that constrain the range of values that a particular field within a pattern may have. The are two types of field constraints: logical operations and predicate functions [CUL89]. The logical operation field constraint is the only one that used in the Graptool program.

There are three types of logical operators: the & representing AND operator, | representing OR operator, and ~ representing NOT operator. All three logical operators can be combined almost in any form in a matching pattern. Evaluation of multiple logicals occur from left to right. The syntax of the three logical operators are the following [CUL89]:

<value1>&<value2>	;the AND operator
<value1>|<value2>	;the OR operator
~<value>	;the NOT operator

The AND operator can be used with variable bindings as in the following [CUL89]:

?x&<value1>|<value2>	;the OR operator with variable
?x&~<value>	;the NOT operator with variable

If the field matches the constraints defined by the logical operator, CLIPS will take action based on the number of variable occurrences. If the variable occurs for the first time, the result will be stored in the variable. If the variable has been bound previously, the field must also match the value of the variable.

3.4.5. Constraining patterns

The LHS of a CLIPS rule is a series of relations and needs to be satisfied for the rule to be placed on the agenda. There is no need to define an **explicit and** to the condition because CLIPS assumes that all the rule's patterns are surrounded by it. However, it is possible to define other logical combinations of conditions that cause a rule to perform the actions.

A **logical pattern block** is a collection of LHS patterns that are combined by an **inclusive or** and an **explicit and** logic. The entire block must be satisfied together with all other conditions before the rule performance. A logical block can be combined with other logical blocks in any manner. Each logical block must open and close with a parenthesis. Patterns can also use field constraints. More details about inclusive or, explicit and, and pattern negation are given below.

Inclusive Or

A constraint of **inclusive or** is satisfied when any one of several patterns in an **or** logical block exists. If all other LHS conditions are satisfied, the rule will be activated. The syntax of an **or** logical block is the following [CUL89]:

```
(defrule <name> ["<comment>"]
  [(<additional patterns>)]
    (or (<pattern 1>)
           .
           .
           .
        (<pattern n>))
  [(<additional patterns>)]
    =>
      [(<actions>)])
```

Explicit And

An **explicit and** will allow a logical combination of patterns within an **or** logical block. The constraint is satisfied when all the patterns inside of the **and** logical block match. When all other LHS conditions are satisfied, the rule will be activated. The syntax of an **and** logical block is given [CUL89]:

```
(defrule <name> ["<comment>"]
  [(<additional patterns>)]
   (or (and (<pattern 1>)
                .
                .
                .
            (<pattern n>))
        (<other patterns>))
  [(<additional patterns>)]
    =>
      [(<actions>)])
```

Pattern Negation

A **not** function provides the capability to fire a rule when a fact does not exist in the fact base. The syntax of pattern negation is the following [CUL89]:

```
(defrule <name> ["<comment>"]
      [(<preceding patterns>)]
          (not (<pattern 2>))
      [(<additional patterns>)]
      =>
              [(<actions>)])
```

3.5. *The CLIPS Right Hand Side Rule Syntax -- Actions*

The right-hand side (RHS) is a list of actions that will be performed sequentially when all the conditions (LHS) of the rule are satisfied. However, only **assert** (adding new facts into the fact base) and **retract** (removing facts from fact base) have been used in the Graptool software and will be explained below.

3.5.1. Add new facts to fact base

New facts can be added to the fact list by the CLIPS **assert** command. A new fact will not be added if that fact already exists in the fact list. An assert can be used only on the right hand side of the rule. The syntax of assert is the following [CUL89]:

(assert (<pattern>)[(<additional patterns>)])

3.5.2. Remove facts from fact base

The **retract** command will remove facts from the fact base. Facts can be identified as a **fact variable** (?<fact_var>) bound on the LHS of the rule. The syntax of retract is the following [CUL89]:

(retract ?<fact-var> [?<fact-var>] [?<fact-var>] [?<fact-var>])

3.6. *The CLIPS Commenting Rules*

In CLIPS, a comment can directly follow the name of **defrule**, **deffacts**, and **deftemplate**. The comment can also be anywhere in CLIPS after a semicolon (;). Everything from the semicolon to the next carriage return (linefeed) will be ignored.

4. LOGICAL PATH GRAPH ALGORITHM

Before the actual logical path graph algorithm is given, the reasons that the logical path graph method was chosen as a testing method for rule-based ES will be described. The following discussion is taken from [KIPER92].

A logical path graph method describes accurately the appropriate representation of a rule base to identify the logical flow of data through it. Using this representation, various data flow test path selection techniques can be used to obtain sets of test paths that satisfy appropriate adequacy criteria. This method was chosen over other methods of testing (physical rule flow graphs and causality graphs) for a couple of reasons. First, physical rule flow graphs are dependent on knowledge of the inference engine and a rule base's physical order of execution is generally not the same as its logical order of execution. Secondly, a causality graph does not capture all of the logical paths through the rule base.

The **logical path graph** satisfies the three test method criteria for a rule-based expert system: inference engine independence, adequate coverage, and computational effectiveness. It is independent of the physical order of rule firing as determined by the inference engine. It is also semantically equivalent to that of a control flow graph from traditional software where a rule forms the basis for a node and two nodes are connected by an edge if and only if the second node can follow the first under some set of conditions. Computational effectiveness is determined by the relationship between the number of rules and the number of basis paths. The tentative experimental evidence indicates that the size of the graph and the order of the algorithm are generally reasonable enough to make this method practical.

In order for the logical path graph to be strongly connected, there are four conditions that would need to be satisfied. The first condition requires that each rule be reachable. Since it is possible to have unsatisfiable rules on the LHS, such rules would never fire and would have to be eliminated. The second condition requires that there be an addition of a special initial node to the graph that would be connected to all rules that can be fired as the initial action of the inference engine. Since every node can be reached from the initial node, every rule will ultimately fire as a result of some combination of initial and input values. The third condition is that each graph would have a terminal node. This means that any node representing the last node in a logical path would be connected to a terminal node. The fourth condition requires that an edge is added to connect the terminal node to the initial node. Since every node can be reached from the initial node, every node can reach the terminal node; and since the terminal node is connected to the initial node, the graph is strongly connected.

Logical path graph construction algorithm

Assumptions: Every rule is given a unique number and all unreachable rules have been eliminated from the rule base.

Basic concepts of logical path graph:

1) A particular rule may be represented by more than one node.

2) Each node is uniquely labeled using two pieces of information: the number of the associated rule, and a **condition set** (set of all conditions asserted by nodes on the path leading to the node). The condition set for a node is the cumulative effect of all assertions and retractions that have occurred along this path [KIPER92].

3) Rules represented as Ri where index *i* is the unique number associated with the *ith* rule.

4) Any node corresponding to rule Ri is called Nij and the associated condition set Cij

5) Since any rule may be represented as more than one node, two indices are needed to label the node and condition set. The first index *i* indicates that this condition set is associated with rule Ri. The second index *j* means that this is the *jth* node and condition set associated with rule *i* [KIPER92].

Construction algorithm:

Step 1.: Create the start node $N_{0,1}$ associated with no rule and give its condition set $C_{0,1}$ the value $\emptyset$, the empty set. Also, create a special terminating node called $N_{f,1}$ with $C_{f,1} = \emptyset$

Step 2.: For any rule R_j with a vacuous LHS, or whose LHS consists entirely of the initial fact, create a node $N_{i,1}$ with $C_{i,j} = \emptyset$. Connect $N_{0,1}$ to each such $N_{i,1}$ with an edge.

Step 3.: For any rule R_m whose LHS conditions are satisfied by the RHS conditions of some rule R_i corresponding to an existing node $N_{i,j}$ plus its associated condition set $C_{i,j}$, create a new node $N_{m,p}$. The condition set $C_{m,p}$ is defined to be $C_{i,j}$ union the assertions of R_i minus the retractions of R_i. Add an edge from $N_{i,j}$ to $N_{m,p}$. If such a node $N_{m,p}$, $C_{m,p}$ is identical to an existing node and condition set, then do not add a new node, but insert a new edge from $N_{i,j}$ to $N_{m,p}$ (unless this edge already exists also.)

```
Step 3.: extensions
 repeat
  for i = 1 to $N_f$ do
  begin
    for k in 1 to $n_j$ do
    begin
      Step 3a.: Consider all combinations of k nodes whose RHS conditions plus associated
          condition sets satisfy the LHS conditions of $R_j$ while no proper subset of this
          combination satisfies these LHS conditions. For each such combination, add a new
          node $N_{i,j}$ whose condition set is the union of the k condition sets union the
          assertions of $R_i$ minus the retractions of $R_i$, and where j is defined as an integer one
          greater than the largest x such that $C_{i,x}$ is an existing condition set. If $k > 1$, these
          rules form an "AND" group. If this pair of new node and condition set are
          identical to an existing pair, then do not create a new node, rather add an edge.
    Step 3b.: Connect this new node to all the nodes corresponding to the condition sets in
        the combination of k nodes from step 3a.  If k > 1, use the AND notation on the
        incoming edges to denote this.
    Step 3c.: If any of these antecedent rules is not connected to the terminating node, and
        is independent from at least one other rule in the condition set, and there is no other
        edge leading from this node, then add an edge connecting this node to the
        terminating node.
  end for k
 end for i
until no new nodes or edges are added to the graph
Step 4.: For every node that has no edges leading out from it, add an edge to node
   Nf.[KIPER92].
```
The above algorithm will be modified to be used in the Graptool program.

5. GRAPTOOL PROGRAM

Originally, this program was named *Graphtool* because it is a computer tool which has been developed based on the **logical path graph algorithm**. However, the DOS operating system does not allow any file name to be more than eight characters long. The word *Graphtool* has nine characters. Therefore, the character **h** was taken out from the word *Graphtool* to make the eight character requirement. The name of this software then has become **Graptool**.

Program objective

Graptool was designed as a software tool which would be able to use the logical path graph algorithm for the structural testing of a rule-based expert system. The expert system that will be used with Graptool is called CLIPS. Graptool reads a CLIPS rule base and converts it into Graptool format. After all the conversions, Graptool applies the logical path graph algorithm to the Graptool format rule bases for testing the CLIPS rule-based structure. The results include the following:

1) Graptool.wrk file which contains the CLIPS rule bases after standardized processing.
2) Graptool.rul file which contains CLIPS rules in Graptool format.
3) Graptool.fac file which contains CLIPS initial facts in Graptool format.
4) Graptool.err file (optional) which contains process and error messages during software execution.
5) An information file which contains results of the rule-based structural testing.

Graptool functions

Graptool software was divided into four sections: initialization section, conversion section, preparation section, and an application section. A brief description and outline of each section follows:

The ***initialization section*** will prepare the end user and Graptool software itself for program processing. Functions in this section include the following:

1) To display a brief description of the Graptool software to the end user.
2) To initialize all the important variables.
3) To set up the working directory.
4) To open the CLIPS rule-based file, the error file (*Graptool.err),* the working files (*Graptool.fac*, *Graptool.rul*, and *Graptool.wrk*), and the information file.

The functions involved in this section include the following:

a) **void ProgramIntroduction(void)**

b) **void InitialValue(void)**

c) **void SetWorkingDirectory(char*)**

d) **void OpenTheFiles(char*)**

The ***conversion section*** is used to standardize and convert the CLIPS rule bases and initial facts into Graptool format. Functions in this section include the following:

1) To standardize the CLIPS rule bases to become the same format.
2) To check for possible critical errors in the CLIPS rule bases.
3) To convert the CLIPS rules and initial facts into Graptool format.

The functions involved in this section include the following:

a) **void ReadTheRulebaseInToWorkingFile(void)**

b) **int ReadRuleAndFactFromWorkingFile(int)**

c) **void ConvertRulebaseToGraptoolFormat(int)**

The ***preparation section*** will prepare the initial facts and rule bases for rule-based structural testing. Functions in this section include the following:

1) To read the initial facts from the *Graptool.fac* file into the ***fact_base*** array. If initial facts do not exist in *Graptool.fac*, the user will be asked to enter initial facts.
2) To read the Graptool format rule bases from the *Graptool.rul* into a ***rule_base*** array.

The functions involved in this section include the following:

a) **void PrepareInitialFacts(void)**

b) **void ReadRulebaseFromWorkingFile(void)**

The ***application section*** will apply the logical path algorithm to test the rule-based structure. Functions in this section include the following:

1) To search the ***rule_base*** array for a rule whose conditions have been satisfied by the fact base. This rule is called a **working rule.**
2) To generate the new node and its condition set.
3) To display the list of node connections and store that list in the selected information file.
4) To do simple rule-based analysis.

The functions involved in this section include the following:

a) **char *SearchForTheWorkingRule(int*)**
b) **void AssertNewFact(char*)**
c) **void RetractOldFact(char*)**
d) **void NodeGenerator(void)**
e) **void DisplayTestResult(void)**
f) **void FinalAnalysis (void)**

Any more details for each function can be found in appendix b.

Extra procedures

In order to understand the Graptool program, two extra procedures have been added to make the program execute smoother and easier to use. These two extra procedures are the counting method and the Graptool format.

Counting method

The counting method was used for identifying the location of each character in the rule base. This method has been used frequently in the **<u>ReadTheRulebaseInToWorkingFile</u>** and **<u>ReadRuleAndFactFromWorkingFile</u>** functions. The counting method works in two ways as illustrated in the figure below:

```
1
(defrule rule one
        2                             1
  ?x <- (condition-one "testing ?y" ?z)
=>
 2       3             21
 (assert (condition-two))
 2          10
 (retract ?x))
```

Figure 1. Counting method.

The counting method can recognize a character that is inside a **block** by first assigning zero to a variable called parenthesis counter which is identified as the parenthes_count variable inside the **ReadTheRulebaseInToWorkingFile** function and the **ReadRuleAndFactFromWorkingFile** function. Graptool will add one to the parenthesis counter each time it finds an open parenthesis and subtract one each time it finds a closed parenthesis. By doing this, Graptool can identify the position of each character relative to a **block**. In the figure 1 above, the question mark in front of x has a number of one in the parenthesis counter, which means that it is a part of the **retract index**. The question mark in front of z has a number of two which means that it is part of a single field variable. The equal sign has a number of one in the parenthesis counter which means that it is a part of an CLIPS **arrow**. The parenthesis counter will become zero again at the end of each rule base.

The counting method can recognize a character within quotation marks (**string field**) by first assigning zero to a variable called quote counter which is identified as quote_count variable. Graptool will add one to the quote counter each time it finds a quotation mark. In order to determine whether or not a character is inside quotation marks, Graptool will divide the quote counter by two. If the dividing remainder is zero, it means that a character is not inside quotation marks. If the remainder is one, it means that the character is inside quotation marks. However, if the character is a quotation mark and the remainder is one, then that quotation mark is the beginning of the **string field**. Similarly, if the character is a quotation mark and the remainder is zero, then that quotation mark is the end of the **string field**.

Graptool format

One of the purposes for developing Graptool is to test the rule-based structure of a large collection of rule bases. Since computer memory is limited, Graptool format has been created to save memory. The Graptool format procedures are the following:

1. Graptool will replace the CLIPS **defrule**, **deffacts**, **assert**, **retract**, and **arrow** (=>) with a single unprintable character. The unprintable characters are DEFRULE, DEFFACTS, CONDITION, ASSERT, RETRACT, and LHSRHS.
2. In the Graptool format, a space is used as the separation between the **fields** inside the **block**. An open and closed parentheses is used as the beginning and end of a **block**. Therefore, Graptool will replace a space, an open parenthesis, and a closed parenthesis with Q_SPACE, O_PAREN, and C_PAREN characters, respectively.
3. The CLIPS command *and*, *or*, and *not* in the **logical pattern block** will be replaced by &, | , and ~ , respectively.
4. The end of the rule will be identified by the ENDRF character and the end of the initial fact will be identified by the ENDARRAY character.

For more details about any of these replaced characters, see appendix a.

```
DEFRULE (rule one) CONDITION&(condition-one "testing ?y" ?z)
LHSRHS
ASSERT(condition-two) RETRACT(condition-one "testing ?y" ?z) ENDRF
```

Figure 2. Graptool format.

Figure 2 above is the previous figure 1 in Graptool format.

6. INSTRUCTIONS FOR USING GRAPTOOL SOFTWARE

Graptool is a user friendly software which contains easy instructions for the end user to follow. Although there are some limitations which will be discussed later, Graptool can handle the following without modification: **LHS literal patterns, wildcards (single** and **multifield), variable (single** and **multifield), constraining fields, logical operators,** and **constraining patterns.** For the RHS, Graptool does the **retract** and **assert.** Anything else in the RHS will be ignored by Graptool. Examples of how to use Graptool are shown through execution of the *block.clp* CLIPS rule base file in [RIL89], pages 423-24.

The objective of *block.clp* rule base is to rearrange the stack of blocks into a gold configuration with the minimum of moves. The *block.clp* rule base can be modified into *block1.clp*. In *block1.clp*, the print statements (**printout t...**) from *block.clp* were changed into assert fact statements. The source code of the *block.clp* is shown below together with the *block1.clp*.

```
(deffacts initial-state
  (stack A B C)
  (stack D E F)
  (move-goal C on-top-of E)
  (stack))

(defrule move-directly
  ?goal <- (move-goal ?block1 on-top-of ?block2)
  ?stack-1 <- (stack ?block1 $?rest1)
  ?stack-2 <- (stack ?block2 $?rest2)
  =>
  (retract ?goal ?stack-1 ?stack-2)
  (assert (stack $?rest1))
  (assert (stack ?block1 ?block2 $?rest2))
  (printout t ?block1 " move on top of " ?block2 "." crlf))
```

BLOCK.CLP SOURCE CODE

```
(deffacts initial-state
  (stack A B C)
  (stack D E F)
  (move-goal C on-top-of E)
  (stack))

(defrule move-directly
  ?goal <- (move-goal ?block1 on-top-of ?block2)
  ?stack-1 <- (stack ?block1 $?rest1)
  ?stack-2 <- (stack ?block2 $?rest2)
  =>
  (retract ?goal ?stack-1 ?stack-2)
  (assert (stack $?rest1))
  (assert (stack ?block1 ?block2 $?rest2))
  (assert (printout t ?block1 " move on top of " ?block2 "." crlf)))
```

BLOCK1.CLP SOURCE CODE

```
(defrule move-to-floor
  ?goal <- (move-goal ?block1 on-top-of floor)
  ?stack-1 <- (stack ?block1 $?rest)
  =>
  (retract ?goal ?stack-1)
  (assert (stack ?block1))
  (assert (stack $?rest))
  (printout t ?block1 " move on top of floor." crlf))

(defrule clear-upper-block
  (move-goal ?block1 on-top-of ?)
  (stack ?top $? ?block1 $?)
  =>
  (assert (move-goal ?top on-top-of floor)))

(defrule clear-lower-block
  (move-goal ? on-top-of ?block1)
  (stack ?top $? ?block1 $?)
  =>
  (assert (move-goal ?top on-top-of floor)))
```

BLOCK.CLP SOURCE CODE

```
(defrule move-to-floor
  ?goal <- (move-goal ?block1 on-top-of floor)
  ?stack-1 <- (stack ?block1 $?rest)
  =>
  (retract ?goal ?stack-1)
  (assert (stack ?block1))
  (assert (stack $?rest))
  (assert (printout t ?block1 " move on top of floor." crlf)))

(defrule clear-upper-block
  (move-goal ?block1 on-top-of ?)
  (stack ?top $? ?block1 $?)
  =>
  (assert (move-goal ?top on-top-of floor)))

(defrule clear-lower-block
  (move-goal ? on-top-of ?block1)
  (stack ?top $? ?block1 $?)
  =>
  (assert (move-goal ?top on-top-of floor)))
```

BLOCK1.CLP SOURCE CODE

An example of the block1.clp sequence of execution is shown below with the sequence of screens that the end user will encounter during Graptool execution. These screens will have a sample end user response when it is required and will move onto the next screen resulting from that particular response. The user response will be inserted in **bold** characters. The first screen is the information screen which will continue to be displayed until the end user presses a key:

```
================================================================
                    ******** GRAPTOOL.C ********
PROGRAM OBJECTIVE.: Graptool is a software tool based on the logical path graph
algorithm. This software will read the CLIPS rule base and convert it into
Graptool format. After the conversion is finished, the program will apply the
logical path graph algorithm to the Graptool format for testing of rule-based
structure. The results of this tool are the following:
1. GRAPTOOL.WRK contains the CLIPS rule base after Graptool has eliminated the
   unnecessary functions or commands from the original rule base.
2. GRAPTOOL.RUL contains rules from the rule base in Graptool format.
3. GRAPTOOL.FAC contains initial facts from the rule base in Graptool format.
4. GRAPTOOL.ERR (option) contains process and error messages during software
   execution.
5. The information file contains the results of rule base testing. An user can
   select any file name except the CLIPS rule base file name, and the file
   extension cannot be '.CLP'. If the end user makes an invalid name selection,
   Graptool will select a unique file name which starts with 'I'.

FINAL NOTE: All five files will be saved in the selected working directory.
WARNING..: COMPUTER WILL AUTOMATICALLY STOP EXECUTION IF IT FIND ANY ERROR.
================================================================

Press any key to continue........
```

```
* THE CURRENT WORKING DIRECTORY -> C:\

Do you want to change the working directory (N)?  Y
```

```
!! MAXIMUM PATH INCLUDING FILE NAME IS 30 BYTES.
Enter the drive of new working directory (A, B, C, etc.):  C
Enter the path of new working directory (\ for root):  BORLANDC\PROGRAM
* THE CURRENT WORKING DIRECTORY -> C:\BORLANDC\PROGRAM

Do you want to change the working directory (N)?  N
```

```
Enter the name of CLIPS rule base file ->  BLOCK1.CLP
Enter the name of information file (option) ->  BLOCK1.INF
Do you want to open the error file (N)?  Y
```

```
**************** PROGRAM SETUP ***************
 The fact string size is 80 bytes.
 The field string size is 60 bytes.
 The rule array size is 1024 bytes.
 The fact_base array size is 500 bytes.
 The variable array size is 2000 bytes.
 The selected working directory is C:\BORLANDC\PROGRAM
 The CLIPS rule base file is BLOCK1.CLP
 The information file is BLOCK1.INF
 The error file is GRAPTOOL.ERR.
**************************************************

Press any key to continue..........
```

The following is an execution screen which will show the sequence of conversion of CLIPS rule base into Graptool format. It will show the name of each initial fact and rule, and provides some information about reading rules and facts. This information includes error checking messages and conversion messages. At the end of this screen, the computer will ask if the end user wants to add any extra facts to the fact base. The response, in this case, is yes.

```
_-_ Read CLIPS rule base into the working file _-_

_-_-_ Complete the reading _-_-_


** Converting rule bases into Graptool format.

FINISH READING.....deffacts initial-state
** DOES NOT DETECT ANY ERROR IN READING PROCESS **
Convert deffacts to Graptool format........
```

```
FINISH READING.....defrule move-directly
** DOES NOT DETECT ANY ERROR IN READING PROCESS **
Converting condition to Graptool format...
Convert assert and retract to Graptool format...

FINISH READING.....defrule move-to-floor
** DOES NOT DETECT ANY ERROR IN READING PROCESS **
Converting condition to Graptool format...
Convert assert and retract to Graptool format...

FINISH READING.....defrule clear-upper-block
** DOES NOT DETECT ANY ERROR IN READING PROCESS **
Converting condition to Graptool format...
Convert assert and retract to Graptool format...
!!!!!!! NO RETRACT IN THIS RULE. !!!!!!!!!

FINISH READING.....defrule clear-lower-block
** DOES NOT DETECT ANY ERROR IN READING PROCESS **
Converting condition to Graptool format...
Convert assert and retract to Graptool format...
!!!!!!! NO RETRACT IN THIS RULE. !!!!!!!!!

** Converting process is finished.

<<<< Start reading facts from Graptool.fac >>>>

FINISH READING DEFFACTS...initial-state

<<<< End reading facts from the Graptool.fac >>>>

Do you want to add the extra facts to fact base (N)?  Y
```

This is the entering of extra facts screen. The first extra fact entered is an **(initial-fact)**. The computer will not accept this fact because it already exists inside the fact base. The reason for this is that Graptool automatically adds an **(initial-fact)** to the fact base because *block1.clp* rule base contains the **deffacts construct**.

```
****************** IMPORTANT INSTRUCTIONS ********************
 The size of the fact_base array is 500 bytes.
 EVERY INITIAL FACT MUST START AND END WITH A PARENTHESES.
 Otherwise the computer will not accept the given fact.
 PRESS RETURN (ENTER) AFTER FINISHING INPUT A INITIAL FACT.
 Otherwise the computer will not start the fact process.
 The computer will accept ONLY ONE INITIAL FACT AT A TIME.
 The syntax of the fact is the SAME as the fact of CLIPS.
*****************************************************************

!! FACT STRING CANNOT BE LONGER THAN 80 BYTES. !!
!! EACH FIELD CANNOT BE LONGER THAN 60 BYTES.  !!
** You have 426 bytes left in the fact_base array. **
Enter the fact string ........
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
(initial-fact)
Processing initial fact ........

ALREADY EXISTS. <!!>
```

The second extra fact entered is a **(block red | blue)**. The computer does not accept this fact because it contains a logical operator **or**.

```
!! FACT STRING CANNOT BE LONGER THAN 80 BYTES. !!
!! EACH FIELD CANNOT BE LONGER THAN 60 BYTES.  !!
** You have 426 bytes left in the fact_base array. **
Enter the fact string ........
1234567890123456789012345678901234567890123456789012345678901234567890123456789
(block red | blue)
Processing initial fact ........

ERROR! GIVEN FACT HAS SYNTAX ERROR.
```

The last extra fact entered is **(block red)**. The computer accepts this fact because each field is valid (if **word**, **number**, or **string**) and does not duplicate any fact in the fact base. The **(block red)** fact was added to show that Graptool allows the end user to add extra facts to the fact base. However, this fact does not affect the result of rule-based structural testing because it will not be used by any rule.

```
!! FACT STRING CANNOT BE LONGER THAN 80 BYTES. !!
!! EACH FIELD CANNOT BE LONGER THAN 60 BYTES.  !!
** You have 426 bytes left in the fact_base array. **
Enter the fact string ........
1234567890123456789012345678901234567890123456789012345678901234567890123456789
(block red)
Processing initial fact ........

HAS BEEN ACCEPTED. <**>
```

If the end user does not enter an open parenthesis, the computer will assume that the end user wants to stop entering extra facts. After that, the computer will ask the end user to confirm the entering of extra facts. If the end user enters Y, the computer will assume that the end user wants to re-enter every fact again. In this example, N was entered.

```
!! FACT STRING CANNOT BE LONGER THAN 80 BYTES. !!
!! EACH FIELD CANNOT BE LONGER THAN 60 BYTES.  !!
** You have 415 bytes left in the fact_base array. **
Enter the fact string ........
1234567890123456789012345678901234567890123456789012345678901234567890123456789
↵

@@@ ALL INPUT FACTS WILL BE DELETED IF YOU SELECT 'Y'. @@@
Do you want to re-enter the facts in fact base (N)?  N
```

To begin the rule-based structural testing requires the end user to enter Y. This is also the last chance for the end user to stop the testing process by entering anything besides Y or y.

```
!!***!! EVERYTHING IS READY FOR TESTING. !!***!!

** Enter Y to start the rule-based testing process..> Y
```

The end user has the option of changing the order of retraction and assertion of facts. The default selection involves the assertion of facts to the fact base before retraction. Entering N will result in retraction before assertion.

```
Enter N if you want (FACT BASE - RETRACT U ASSERT),
or anything else (FACT BASE U ASSERT - RETRACT)...> Y
```

The rest of these screens are the result of rule-based structural testing by the logical path graph algorithm.

```
*** Starting rule base test ***

Rule number 0 is the initial-state.
Rule number 1 is the move-directly.
Rule number 2 is the move-to-floor.
Rule number 3 is the clear-upper-block.
Rule number 4 is the clear-lower-block.

Working NODE(0, 0)
It's condition set is...
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
...and connect to the following nodes:

NODE(3, 0)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal A on-top-of floor)
```

```
NODE(4, 0)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)

Working NODE(3, 0)
It's condition set is...
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal A on-top-of floor)
...and connect to the following nodes:

NODE(2, 0)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)

NODE(3, 0)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal A on-top-of floor)

NODE(4, 1)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal A on-top-of floor)
(move-goal D on-top-of floor)
```

```
Working NODE(4, 0)
It's condition set is...
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
...and connect to the following nodes:

NODE(2, 1)
(stack A B C)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)

NODE(3, 1)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
(move-goal A on-top-of floor)

NODE(4, 0)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)

Working NODE(2, 0)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
...and connect to the following nodes:
```

```
NODE(3, 2)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)

NODE(4, 2)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)

Working NODE(4, 1)
It's condition set is...
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal A on-top-of floor)
(move-goal D on-top-of floor)
...and connect to the following nodes:

NODE(2, 2)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)

NODE(3, 1)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal A on-top-of floor)
```

```
(move-goal D on-top-of floor)

NODE(4, 1)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal A on-top-of floor)
(move-goal D on-top-of floor)

Working NODE(2, 1)
It's condition set is...
(stack A B C)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
...and connect to the following nodes:

NODE(3, 3)
(stack A B C)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(move-goal A on-top-of floor)

Working NODE(3, 1)
It's condition set is...
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
(move-goal A on-top-of floor)
...and connect to the following nodes:

NODE(2, 3)
(stack A B C)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
```

```
(move-goal A on-top-of floor)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)

NODE(3, 1)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
(move-goal A on-top-of floor)

NODE(4, 1)
(stack A B C)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
(move-goal A on-top-of floor)

Working NODE(3, 2)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
...and connect to the following nodes:

NODE(2, 4)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)

NODE(3, 2)
(stack D E F)
(move-goal C on-top-of E)
```

```
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)

NODE(4, 3)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
(move-goal D on-top-of floor)

Working NODE(4, 2)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)
...and connect to the following nodes:

NODE(2, 5)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)

NODE(3, 4)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
```

```
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)
(move-goal B on-top-of floor)

NODE(4, 2)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)

Working NODE(2, 2)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
...and connect to the following nodes:

NODE(2, 5)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)

NODE(3, 4)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
```

```
NODE(4, 2)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal D on-top-of floor)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)

Working NODE(3, 3)
It's condition set is...
(stack A B C)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(move-goal A on-top-of floor)
...and connect to the following nodes:

NODE(2, 5)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)

NODE(3, 3)
(stack A B C)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(move-goal A on-top-of floor)

Working NODE(2, 3)
It's condition set is...
(stack A B C)
(move-goal C on-top-of E)
(stack)
(initial-fact)
```

```
(block red)
(move-goal A on-top-of floor)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
...and connect to the following nodes:

NODE(2, 5)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)

NODE(3, 3)
(stack A B C)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(move-goal A on-top-of floor)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)

Working NODE(2, 4)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)
...and connect to the following nodes:

NODE(4, 4)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
```

```
(stack C)
(printout t B "move on top of floor." crlf)
(move-goal D on-top-of floor)

Working NODE(4, 3)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
(move-goal D on-top-of floor)
...and connect to the following nodes:

NODE(2, 6)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)

NODE(3, 4)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
(move-goal D on-top-of floor)

NODE(4, 3)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
```

```
(move-goal D on-top-of floor)

Working NODE(2, 5)
It's condition set is...
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
...and connect to the following nodes:

NODE(3, 5)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(move-goal B on-top-of floor)

Working NODE(3, 4)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)
(move-goal B on-top-of floor)
...and connect to the following nodes:

NODE(2, 7)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
(stack D)
```

```
(stack E F)
(printout t D "move on top of floor." crlf)

NODE(3, 4)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)
(move-goal B on-top-of floor)

NODE(4, 3)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)
(move-goal B on-top-of floor)

Working NODE(4, 4)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)
(move-goal D on-top-of floor)
...and connect to the following nodes:

NODE(2, 8)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)
(stack D)
```

```
(stack E F)
(printout t D "move on top of floor." crlf)

NODE(4, 4)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)
(move-goal D on-top-of floor)

Working NODE(2, 6)
It's condition set is...
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(move-goal D on-top-of floor)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)
...and connect to the following nodes:

NODE(2, 8)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)

NODE(4, 4)
(stack D E F)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
```

```
(move-goal D on-top-of floor)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)

Working NODE(3, 5)
It's condition set is...
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(move-goal B on-top-of floor)
...and connect to the following nodes:

NODE(2, 8)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)

NODE(3, 5)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(move-goal B on-top-of floor)

Working NODE(2, 7)
It's condition set is...
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
```

```
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
...and connect to the following nodes:

NODE(2, 8)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)

NODE(3, 5)
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(stack B C)
(printout t A "move on top of floor." crlf)
(move-goal B on-top-of floor)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)

Working NODE(2, 8)
It's condition set is...
(move-goal C on-top-of E)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(stack C)
(printout t B "move on top of floor." crlf)
(stack D)
(stack E F)
(printout t D "move on top of floor." crlf)
...and connect to the following nodes:
```

```
NODE(1, 0)
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(printout t B "move on top of floor." crlf)
(stack D)
(printout t D "move on top of floor." crlf)
(stack C E F)
(printout t C "move on top of" E "." crlf)

Working NODE(1, 0)
It's condition set is...
(stack)
(initial-fact)
(block red)
(stack A)
(printout t A "move on top of floor." crlf)
(stack B)
(printout t B "move on top of floor." crlf)
(stack D)
(printout t D "move on top of floor." crlf)
(stack C E F)
(printout t C "move on top of" E "." crlf)
...and connect to the following nodes:


TERMINATION NODE

*** Rule base test is finished ***

********** Rulebase Structural Analysis **********

RULE NUMBER - 0
NUMBER OF NODES - 1
NUMBER OF DUPLICATE NODES - 0

RULE NUMBER - 1
NUMBER OF NODES - 1
NUMBER OF DUPLICATE NODES - 0

RULE NUMBER - 2
NUMBER OF NODES - 15
NUMBER OF DUPLICATE NODES - 6

RULE NUMBER - 3
NUMBER OF NODES - 17
NUMBER OF DUPLICATE NODES - 11
```

```
RULE NUMBER - 4
NUMBER OF NODES - 14
NUMBER OF DUPLICATE NODES - 9

*************************************************
```

Rulebase modifications

Even though Graptool can handle **constraining patterns**, it will not handle a complicated **logical pattern block**. Graptool allows each rule to have three logical blocks, **inclusive or**, **pattern negation**, and **explicit and**. Each logical block cannot have logical sub-blocks. Each logical operator is connected by the AND logical operator. For example, the left hand side logical block is the **inclusive or** block and has no logical sub-block. See figure 3 below. Graptool has no problem with this **logical pattern block**.

```
(or (light red)
    (walk-sign don't walk)
    (police say don't walk))
```

Figure 3. **Inclusive or** block.

The logical block shown below is also the **inclusive or** block which has three **explicit and** sub-blocks. Graptool will not handle this one. These **or** logical blocks need to be modified. The first block contains *(police say walk)*; the second contains *(walk-sign walk)* and *(light green)*; and the last contains *(walk-sign walk)* and *(light yellow)*.

```
(or (police says walk)
    (and (walk-sign walk)
         (light green))
     (and (walk-sign walk)
          (light yellow)))
```

As another example, these LHS has three logical blocks. **Explicit and** block is *(no car coming)*. **Inclusive or** block contains *(police say walk), (walk-sign walk),* and *(light red),* and the negation block is *(not light green).* All these three blocks are connected with logical AND. Graptool has no problem in processing it.

```
(no car coming)
(or (police says walk)
   (walk-sign walk)
   (light red)))
(not (light-green))
```

In order to modify the LHS **logical pattern block** correctly, the end user needs know the AND-OR expression formula such as $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$. For example:

```
(defrule determine-point-surface state
(query phase)
(or ( and (working-state-engine does-not-start)
(spark-state-engine irregular spark))
(symptom engine low-output))
(not (point-surface-state points))
=>
  ............
```

Assumption..:

$A = \{(\text{query phase})\}$
$B = \{(\text{working-state-engine does-not-start})\}$
$C = \{(\text{spark-state-engine irregular-spark})\}$
$D = \{(\text{symptom engine low-output})\}$
$E = \{\sim(\text{point-surface-state points})\}$

We can express the LHS of Determine-point-surface-state rule as following:

$$\begin{aligned} A \wedge ((B \wedge C) \vee D) \wedge E) &= (A \wedge E) \wedge ((B \wedge C) \vee D)) \\ &= ((A \wedge E) \wedge (B \wedge C)) \vee ((A \wedge E) \wedge D)) \\ &= (A \wedge B \wedge C \wedge E) \vee (A \wedge B \wedge C \wedge D) \end{aligned}$$

From the final expression called the disjunctive normal form, we will separate the rule into two paths. Path one will contain the LHS which is expressed as $(A \wedge B \wedge C \wedge E)$. Another path contains the LHS as expressed in $(A \wedge B \wedge C \wedge D)$. Both paths will have the same RHS. The results are:

```
(defrule determine-point-surface state
(query phase)
(or ( and (working-state-engine does-
not-start)
(spark-state-engine irregular spark))
(symptom engine low-output))
(not (point-surface-state points))
=>
```

Original

```
(defrule      determine-point-surface
state1
(query phase)
(working-state-engine does-not-start)
(spark-state-engine irregular-spark)
(not (point-surface-state points)
=>
```

Path 1

```
(defrule      determine-point-surface
state2
(query phase)
(working-state-engine does-not-start)
(spark-state-engine irregular-spark)
(symptom engine low-output)
=>
............
```

Path 2

The advantage of simplifying the LHS into small paths is that it still maintains the original ability of the rule-based structure, and it also provides the clear result of which path of the rule will be used and performed.

Graptool Limitations

Since Graptool does not handle any type of input by the end user, the rule base needs to be modified in order for Graptool to work. There are two modifications that may solve this input problem. These modifications are the replacement method and the separation method.

```
;--------------------------------
; Program by Chris Ortiz
;--------------------------------
(deffacts star
   (start-fact))

;---------------------------------------------------------------------------
; Check which number to see if it is a factorial? (Ask user)
;---------------------------------------------------------------------------
(defrule start
   ?start <- (start-fact)
=>
  (retract ?start)
  (printout t crlf "Enter a number to see if it is a factorial" crlf)
  (bind ?number (read))
  (assert (factorial ?number 0 ?number)))

;-------------------------------------
; Check number for factorial
;-------------------------------------
(defrule Find_Fact
 ?num <- (factorial ?number ?pass ?start)
 (test (> ?number 1))
=>
 (retract ?num)
 (bind ?pass (+ ?pass 1))
 (assert (factorial =(/ ?number ?pass) ?pass ?start)))

;-----------------------
; Factorial Found
;-----------------------
(defrule Is_Factorial
 ?num <- (factorial 1 ?pass ?start)
=>
 (retract ?num)
 (assert (continue prompt))
 (printout t "Number " ?start " is a factorial " ?pass "!" crlf))
```

Figure 4. CLIPS rule base to be modified.

```
;-----------------------------------
; Prompt User to Continue
;-----------------------------------
(defrule Continue_Prompt
 ?response <- (continue prompt)
=>
 (retract ?response)
 (printout t crlf "Do you wish to continue (y/n)")
 (assert (continue =(read))))

;------------------------------------------------
; Start again if user wishes to continue
;------------------------------------------------
(defrule Go_Again
 ?response <- (continue y)
=>
 (retract ?response)
 (assert (start-fact)))
```

Figure 4. CLIPS rule base to be modified (continued).

Replacement method

The purpose of this method is to replace any set of facts in LHS and RHS with a new fact. This new fact will represent the same thing but will not alter the structure of the rule base. Therefore, the replacement must be done consistently throughout the rule base. For example, in figure 4 above, the rule *Start*, *Find_Fact* and *Is_Factorial* need to be modified. Since there is an endless possibility of numbers that the end user can enter at the rule start, those possible numbers are replaced by **(factorial)** fact. This **(factorial)** fact will replace the *(factorial.....)* fact in *Start*, *Find_Fact*, and *Is_Factorial* rule. Even though ?pass and ?start in the Find_Fact rule does not have values, it does not matter because the value of *?pass* and *?start* are a part of the *(factorial.....)* fact. The result of this changing is shown below:

```
;--------------------------------------------------------------------------
; Check which number to see if it is a factorial? (Ask user)
;--------------------------------------------------------------------------
(defrule start
 ?start <- (start-fact)
=>
 (retract ?start)
 (printout t crlf "Enter a number to see if it is a factorial" crlf)
 (bind ?number (read))
 (assert (factorial)))
```

Figure 5. Replacement.

```
;-------------------------------------
; Check number for factorial
;-------------------------------------
(defrule Find_Fact
  ?num <- (factorial)
  (test (> ?number 1))
=>
  (retract ?num)
  (bind ?pass (+ ?pass 1))
  (assert (factorial)))

;----------------------
; Factorial Found
;----------------------
(defrule Is_Factorial
  ?num <- (factorial)
=>
  (retract ?num)
  (assert (continue prompt))
  (printout t "Number " ?start " is a factorial " ?pass "!" crlf))
```

Figure 5. Replacement (continued).

Separation method

The purpose of the separation method is to divide the rule into two or three parts depending on the number of input choices. For example, in figure 4 above, the *Continue_Prompt* rule will be modified. The *Continue_Prompt* rule asks the end user to enter either ***y*** or ***n*** at a prompt. The result of his input will make the computer assert either *(continue y)* or *(continue n)* to the fact base. Since the main goal of the rule-based structure testing is to determine which rule has an effect on a specific result. Therefore, the *Continue_Prompt* rule will be replaced by two rules named *Continue_Prompt-1/2* and *Continue_Prompt-2/2*. The *Continue_Prompt-1/2* will assert *(continue y)* to the fact base while *Continue_Prompt-2/2* will assert *(continue n)* to the fact base. By using the separation method, the rule *Go_Again* (which initially did not fire) will fire because its conditions are satisfied by the condition set of the node that was generated by rule *Continue_Prompt-1/2*. The node which was created by rule *Go_Again* will cause the rule Start to fire. Since the condition set of the node which was created by rule *Continue_Prompt-2/2* will not satisfy any rule in the rule base, this node will have no node connected to it. The result of this replacing and separating is as follows:

```
;--------------------------------
; Prompt User to Continue
;--------------------------------
(defrule Continue_Prompt
 ?response <- (continue prompt)
=>
 (retract ?response)
 (printout t crlf "Do you wish to continue (y/n)")
 (assert (continue =(read))))
```

before separating

```
;----------------------------------
; Prompt User to Continue path 1
;----------------------------------
(defrule Continue_Prompt-1/2
 ?response <- (continue prompt)
=>
 (retract ?response)
 (printout t crlf "Do you wish to continue (y/n)")
 (assert (continue y)))

;----------------------------------
; Prompt User to Continue path 2
;----------------------------------
(defrule Continue_Prompt-2/2
 ?response <- (continue prompt)
=>
 (retract ?response)
 (printout t crlf "Do you wish to continue (y/n)")
 (assert (continue n)))
```

after separating

Figure 6. Separation method.

The two methods described above are only some suggestions for solving the input problem not handled by Graptool. It may not work in every situation, but it does work in the example above. A creative end user may use similar devices or methods to solve any further problems that may be encountered. The complete modification of the CLIPS rule base (figure 7), the node listing (figure 8), and the logical path graph (figure 9) are shown:

```
;--------------------------------------------------------------------
; Check which number to see if it is a factorial? (Ask user)
;--------------------------------------------------------------------
(defrule start
?start <- (start-fact)
=>
(retract ?start)
(printout t crlf "Enter a number to see if it is a factorial" crlf)
(bind ?number (read))
(assert (factorial)))
```

```
;----------------------------------
; Check number for factorial
;----------------------------------
(defrule Find_Fact
?num <- (factorial)
(test (> ?number 1))
=>
(retract ?num)
(bind ?pass (+ ?pass 1))
(assert (factorial)))

;--------------------
; Factorial Found
;--------------------
(defrule Is_Factorial
?num <- (factorial)
=>
(retract ?num)
(assert (continue prompt)))
(printout t "Number " ?start " is a factorial " ?pass "!" crlf))

;-----------------------------
; Prompt User to Continue
;-----------------------------
(defrule Continue_Prompt-1/2
?response <- (continue prompt)
=>
(retract ?response)
(printout t crlf "Do you wish to continue (y/n)")
(assert (continue y)))

(defrule Continue_Prompt-2/2
?response <- (continue prompt)
=>
(retract ?response)
(printout t crlf "Do you wish to continue (y/n)")
(assert (continue n)))

;-------------------------------------------
; Start again if user wishes to continue
;-------------------------------------------
(defrule Go_Again
?response <- (continue y)
=>
(retract ?response)
(assert (start-fact)))
```
Figure 7. Complete modification of CLIPS rule bases.

```
Rule number 0 is the initial-state.
Rule number 1 is the start.
Rule number 2 is the Find_Fact.
Rule number 3 is the Is_Factorial.
Rule number 4 is the Continue_Prompt-1/2.
Rule number 5 is the Continue_Prompt-2/2.
Rule number 6 is the Go_Again.

FACT BASE MINUS RETRACT UNION ASSERT
```

```
Working NODE(0,0)
It's condition set is...
(start-fact)
(initial-fact)
...and connect to the following nodes:

NODE(1,0)
(initial-fact)
(factorial)

Working NODE(1,0)
It's condition set is...
(initial-fact)
(factorial)
...and connect to the following nodes:

NODE(2,0)
(initial-fact)
(factorial)

NODE(3,0)
(initial-fact)
(continue prompt)

Working NODE(2,0)
It's condition set is...
(initial-fact)
(factorial)
...and connect to the following nodes:

NODE(2,0)
(initial-fact)
(factorial)

NODE(3,0)
(initial-fact)
(continue prompt)

Working NODE(3,0)
It's condition set is...
(initial-fact)
(continue prompt)
...and connect to the following nodes:

NODE(4,0)
(initial-fact)
(continue y)

NODE(5,0)
(initial-fact)
(continue n)
```

```
Working NODE(4,0)
It's condition set is...
(initial-fact)
(continue y)
...and connect to the following nodes:
NODE(6,0)
(initial-fact)
(start-fact)

Working NODE(5,0)
It's condition set is...
(initial-fact)
(continue n)
...and connect to the following nodes:

NO CONNECTING NODE

Working NODE(6,0)
It's condition set is...
(initial-fact)
(start-fact)
...and connect to the following nodes:

NODE(1,0)
(initial-fact)
(factorial)

TERMINATION NODE

***** Rulebase Structural Analysis *****

RULE NUMBER - 0
NUMBER OF NODES - 1
NUMBER OF DUPLICATE NODES - 0

RULE NUMBER - 1
NUMBER OF NODES - 2
NUMBER OF DUPLICATE NODES - 1

RULE NUMBER - 2
NUMBER OF NODES - 2
NUMBER OF DUPLICATE NODES - 1

RULE NUMBER - 3
NUMBER OF NODES - 2
NUMBER OF DUPLICATE NODES - 1

RULE NUMBER - 4
NUMBER OF NODES - 1
NUMBER OF DUPLICATE NODES - 0
```

7. WHAT I LEARNED FROM GRADUATE RESEARCH PROJECT

Technical knowledge:

1. Background on expert system rule bases
2. CLIPS language: how to write a program with it; understand how Graptool does pattern matching, assert, and retract
3. Logical path graph algorithm in testing rule base structure

Personally:

1. I learned how to write a technical paper based on others' research and what I could develop based on the information and concepts from that research. Having to explain many concepts, I learned more formal English that was required for writing and my vocabulary knowledge has expanded.

2. From developing such a large program, I learned how to make a plan for developing it with specific steps. Even though I encountered many problems, I was able to rethink my plan and change my course of action. This resulted in the workable program, Graptool, and gave me a better sense of what I should consider the next time I write a program of this size.

3. I learned the meaning of the term "user friendly." When I first started to develop the Graptool program, I did not think much about it being user friendly until I started to test my program. I found that it was confusing because it did not have good directions. The main problem I encountered was with the working directory because I did not allow the end user to change it. It became difficult when I wanted to change the working directory after I ran the program, so I had it changed. From this problem, I learned that my program not only had to work, but that it also had to be user friendly.

4. When I ran the program in the beginning, it did not work all the time, and determining where the error in my program came from was difficult. I could not tell which function or which section of my program did not work. Therefore I started to add messages to each section in my program to tell which section was beginning to run and when it was finished. Not only did this help me, but it will help anyone using it in the future who might wish to develop Graptool further.

```
RULE NUMBER - 5
NUMBER OF NODES - 1
NUMBER OF DUPLICATE NODES - 0
RULE NUMBER - 6
NUMBER OF NODES - 1
NUMBER OF DUPLICATE NODES - 0

******************************************
```

Figure 8. Node listing of modified CLIPS rule base.

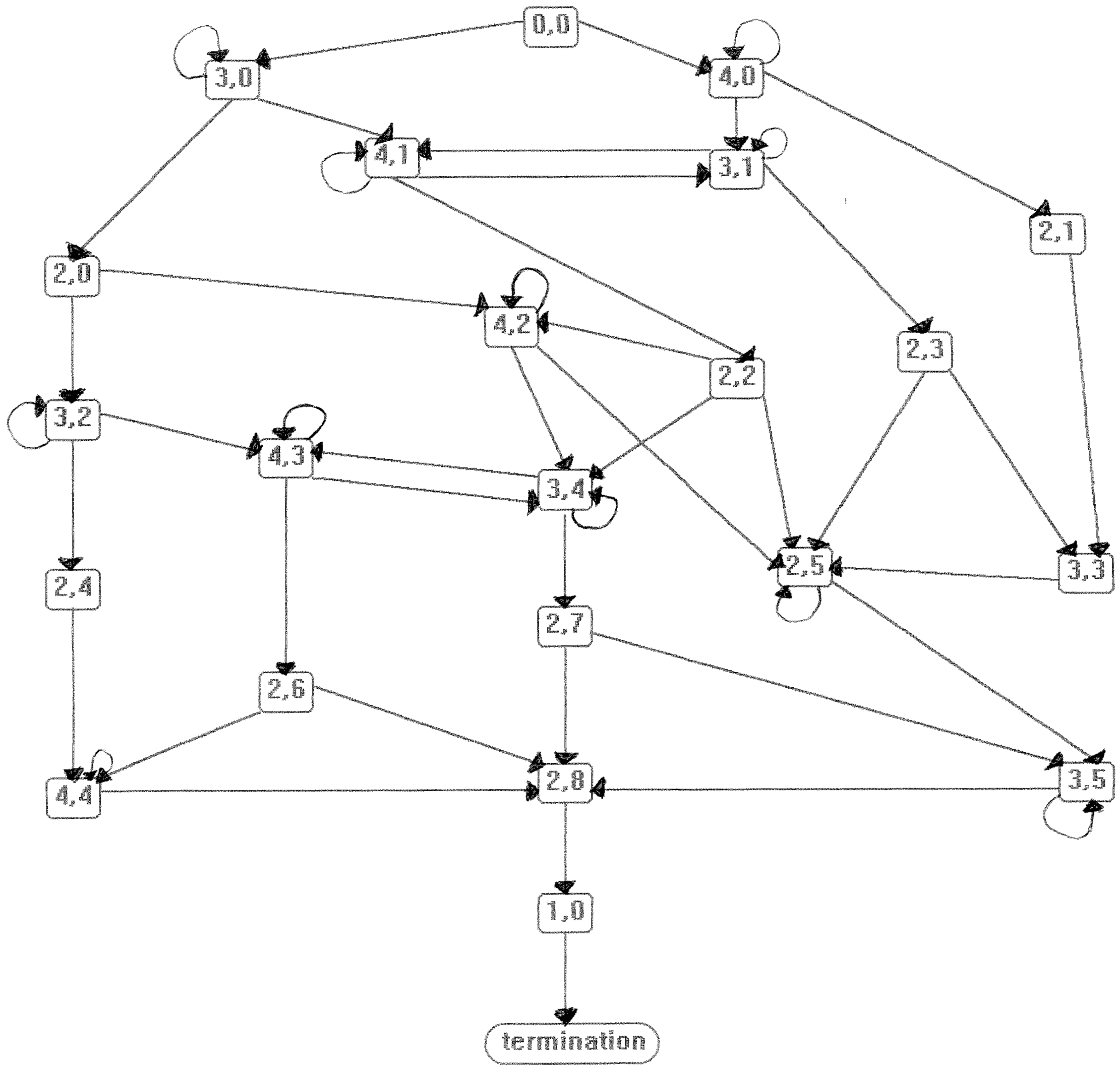


Figure 9. Logical path graph of block.clp.

5. I learned how to use most of the functions of the Microsoft Word for Windows in writing, drawing and refining my paper. I have never used this software before and really enjoyed learning how to use it and how much I could do with it.

8. CONCLUSION

From previous research on the use of graphical representations of rule bases to determine a set of test paths through it, a testing method using the logical path graph algorithm was chosen as the basis from which a computer tool could be developed to create a basis of set of logical paths. A computer tool called Graptool was developed for the purpose of testing the structure of a rule-base in an expert system. Graptool was created using the CLIPS expert system tool and the logical path graph algorithm. Before creating this tool, some background knowledge on expert systems, CLIPS, and structural testing of rule-bases had to be learned. Although Graptool has some limitations already discussed in this paper, it provides an adequate testing of large rule bases in an expert system.

Future work. Graptool's performance may be improved in the following ways:

1. Since modifications had to be made in order for Graptool to handle the CLIPS input command, there are uncertainties as to which cases it can or cannot be applied. Graptool performance might be improved if it allowed the end user more input flexibility.
2. The current result of Graptool produces a node connection listing. There may be more clarity if the results could be put in graphic forms such as the logical path graph of block1.clp in figure 9.
3. Since computer memory is very limited, it cannot handle large node chains. One way to solve this problem would be to install each node and its condition set as a record in the file. This will slow down the rule-based structural testing process, but it will allow the Graptool program to handle larger rule bases.

9. REFERENCES

[KIPER92] Kiper, James D. Ph.D., Structural testing of rule-based expert systems. *ACM Transactions on Software Engineering and Methodology*. 1,2 (1992), 168-187.

[CUL89] Culbert, Chris. *CLIPS Reference Manual*. Artificial Intelligence Section, Lyndon B. Johnson Space Center. NASA, July 1989.

[GIA89] Giarratano, Joseph C. Ph.D. *CLIPS User's Guide*. Artificial Intelligence Section, Lyndon B. Johnson Space Center. NASA, August 1989.

[RIL89] Riley, Gary., Giarratano, Joseph C. *Expert Systems Principles and Programming*. PWS-KENT, Boston, 1989.

[HU89] Hu, David. *C/C++ For Expert Systems*. Management Information Source, Portland, Oregon, 1989.

[LAF90] Lafore, Robert. *The Waite Group's C Programming Using Turbo C++*. SAMS, Carmel, Indiana, 1990.

10. APPENDIX A.: PROGRAM TERMS AND SYMBOLS DEFINITION

This section defines some of the terminology and symbols used throughout the appendix B section. Each term will be written in **bold** letters and each symbol will be written in CAPITAL letters.

TERM.............:	**action block**
DEFINITION.:	A **block** which is in the RHS of **defrule construct**. The action block which Graptool software processes are **assert block** and **retract block**. The action block of the **defrule construct** will be performed if all conditions of that construct are satisfied.
TERM.............:	**arrow** (=>)
DEFINITION.:	A combination of an equal sign (=) with a greater than sign (>) which is used to separate the LHS from the RHS of the rule.
TERM.............:	**assert block**
DEFINITION.:	A **block** in which the **first field** is an **assert** (see section 3.5.1 for more detail). This block also has a **sub-block** right after the **first field**. All variables in **sub-block** must exist in one of the **condition blocks** of the **defrule construct**.
SYMBOL.......:	ASSERT
DEFINITION.:	An ASCII character, number 15, which replaces an **assert** command in Graptool format. A **block** that follows an ASSERT character is called a **new block**.
TERM.............:	**block**
DEFINITION.:	A collection of one or more **fields** that begins with an open parenthesis and ends with a closed parenthesis. There is no limit to the number of **sub-blocks** inside a block.
SYMBOL.......:	COMMENT
DEFINITION.:	An ASCII character, number 59 (;), which indicates the beginning of a CLIPS **comment** (see 3.6 section for more detail).

TERM.............: **condition block** or **pattern**

DEFINITION.: A **block** which is on the LHS of a **defrule construct**. One or more condition blocks can be combined using an **inclusive or** and/or an **explicit and** logic. This combination is called a **logical pattern block** (see section 3.4.5 for more detail). Every pattern must be satisfied by a fact base before the **action block** can be performed.

SYMBOL.......: CONDITION

DEFINITION.: An ASCII character, number 17, which is added in front of each rule base's condition in Graptool format. A **block** that follows a CONDITION character is a condition of the rule base.

SYMBOL.......: C_PAREN

DEFINITION.: An ASCII character, number 22, which will replace a closed parenthesis between the quotes in Graptool format.

SYMBOL.......: C_PARENTHES

DEFINITION.: An ASCII character, number 41 ")", which represents a closed parenthesis. This parenthesis identifies the end of a **block** or **rule-based construct**.

TERM.............: **deffacts construct**

DEFINITION.: A **rule-based construct** in which the **first field** is a **deffacts**. Each block inside a construct is called a **fact block**.

SYMBOL.......: DEFFACTS

DEFINITION.: An ASCII character, number 18, which replaces a **deffacts** command in Graptool format. In the *Graptool.fac* file, a **block** that follows right after a DEFFACTS character is the name of that **deffacts construct**.

TERM.............: **defrule construct**

DEFINITION.: A **rule-based construct** in which the first file is a **defrule** (see section 3.3 for more detail). This construct can be divided into two parts: the LHS and the RHS. A **block** in the LHS is called a **condition block** or a **pattern**. A **block** in the RHS is called an **action block**. The LHS and RHS of a defrule construct is separated by an **arrow**.

SYMBOL.......: DEFRULE

DEFINITION.: An ASCII character, number 14, which replaces a **defrule** command in Graptool format. In the *Graptool.rul* file, a **block** that follows right after a DEFRULE character is the name of that **defrule construct**.

SYMBOL.......: ENDARRAY

DEFINITION.: An ASCII character, number 0 (null), which indicates the end of an array.

SYMBOL.......: ENDLHS and STARTRHS

DEFINITION.: An ENDLHS is an ASCII character, number 61 (=). A STARTRHS is an ASCII character, number 62 (>). The combination of ENDLHS and STARTRHS is an **arrow**.

SYMBOL.......: ENDRF

DEFINITION.: An ASCII character, number 158, which is used to separate the rule bases from one another in Graptool format.

TERM............: **fact block**

DEFINITION.: A **block** which is inside of a **deffacts construct**. Every **field** in a fact block must be either a **word**, a **number**, or a **string**.

TERM............: **field**

DEFINITION.: A single **word**, **string**, **number**, **variable**, or **wildcard** (see section 3.1.2 and 3.4 for more detail) which is separated from one another by a space. The **first field** is a field right after an open parenthesis. The **last field** is a field before a closed parenthesis.

SYMBOL.......: LHSRHS

DEFINITION.: An ASCII character, number 20, which replaces an **arrow** in Graptool format.

SYMBOL.......: LOGI_AND

DEFINITION.: An ASCII character, number 38 (&), which replaces an **and** logical operator of rule patterns in Graptool format.

SYMBOL.......: LOGI_NOT

DEFINITION.: An ASCII character, number 126 (~), which replaces a **not** logical operator of rule patterns in Graptool format.

SYMBOL.......: LOGI_OR
DEFINITION.: An ASCII character, number 124 "|", which replaces an **or** logical operator of rule patterns in Graptool format.

SYMBOL.......: M_WILDCARD
DEFINITION.: An ASCII character, number 36 ($), which identifies a **multifield wildcard** or a **multifield variable** (see section 3.4.2 and 3.4.3 for more detail) when it combines with a S_WILDCARD character.

SYMBOL.......: NEXTLINE
DEFINITION.: An ASCII character, number 10 (linefeed), which causes a computer to go to the next line. The NEXTLINE character also indicates the end of a CLIPS **comment** (see 3.6 section for more detail).

TERM............: **new block**
DEFINITION.: A **block** that follows an ASSERT character. This block will be added to a fact base when all rule base conditions are satisfied. Any variable in a new block must be replaced by its value before a new block is added to a fact base.

SYMBOL.......: O_PAREN
DEFINITION.: An ASCII character, number 21, which replaces an open parenthesis between the quotes in Graptool format.

SYMBOL.......: O_PARENTHES
DEFINITION.: An ASCII character, number 40 "(", which represents an open parenthesis. This parenthesis indicates the beginning of a **block** or **rule-based construct**.

SYMBOL.......: Q_SPACE
DEFINITION.: An ASCII character, number 19, which replaces a space between the quotes in Graptool format.

SYMBOL.......: QUOTE
DEFINITION.: An ASCII character, number 34 ("), which represents a quotation mark.

TERM............: **removing block**

DEFINITION.: A **block** that follows a RETRACT character. This **block** will be removed from a fact base when all conditions of the rule base is satisfied.

SYMBOL.......: RE_POINT1 and RE_POINT2

DEFINITION.: A RE_POINT1 is an ASCII character, number 60 (<). A RE_POINT2 is an ASCII character, number 95 (-). The combination of a RE_POINT1 and a RE_POINT2 is a CLIPS special character (<-).

TERM............: **retract block**

DEFINITION.: A **block** in which the **first field** is a **retract** (see section 3.5.2 for more detail). This block has no **sub-block**. Every **field** beside the **first field** must be a single variable named **fact variable**. All of the fact variables must exist in the LHS of that **defrule construct** and bind a **condition block**.

SYMBOL........: RETRACT

DEFINITION.: An ASCII character, number 16, which replaces a **retract** command in Graptool format. A **block** that follows a RETRACT character is called a **removing block**.

TERM............: **rule-based construct**

DEFINITION.: A collection of **blocks** which opens with a left parenthesis and closes with a right parenthesis. The **first field** of a rule-based construct must be **defrule** or **deffacts** (see section 3.3 for more detail). A second field must be the name of that **rule-based construct**.

SYMBOL.......: SPACE

DEFINITION.: An ASCII character, number 20, which represents a space.

TERM............: **sub-block**

DEFINITION.: A **block** which is inside another block.

SYMBOL.......: S_WILDCARD

DEFINITION.: An ASCII character, number 63 (?), which identifies a **single wildcard** or a **single variable** (see section 3.4.2 and 3.4.3 for more detail).

11. APPENDIX B...: FUNCTION OF GRAPTOOL SOFTWARE

The Graptool software contains thirty four functions including the Main function. All thirty four functions can be divided into two groups: accessory functions and major functions. An explanation of each function follows:

11.1. Accessory functions

There are ten accessory functions in the Graptool software: **Message**, **Error**, **GetAFact**, **GetAField**, **WriteFormatToWorkingFile**, **CheckTheFieldSyntax**, **LogicalOperation**, **FieldUnification**, **FactIsInTheFactbase**, and **ReplaceVariableWithValue**.

1) Message function

Purpose: A **Message** function will display a given message and store it in *Graptool.err* if that file has been opened.

Prototype:

```
void Message(char *first_mess, int mess_number, char *second_mess)
{
```

A first_mess and a second_mess are character pointer variables which contain an address of a message array.

A mess_number is an integer number variable.

Pseudo-code:

Step 1.: If the *first_mess is not an ENDARRAY character (first_mess has an message), the computer will display a message in first_mess on the screen.

```
/* CHECK FOR AN ENDARRAY CHARACTER */
if(*first_mess != ENDARRAY)
 printf("%s", first_mess);
```

Step 2.: If the mess_number is not a NONE (negative one), the computer will display a mess_number value on the screen.

```
/* CHECK FOR A NEGATIVE ONE */
if(mess_number != NONE)
 printf("%d", mess_number);
```

Step 3.: If the second_mess is not an ENDARRAY character (second_mess has an message), the computer will display a message in second_mess on the screen.

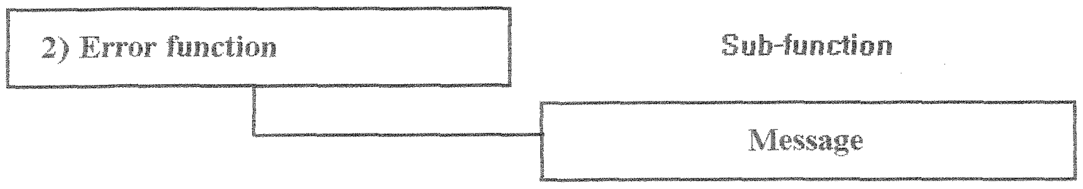
```
/* CHECK FOR AN ENDARRAY CHARACTER */
if(*second_mess != ENDARRAY)
 printf("%s", second_mess);
```

Step 4.: If the *Graptool.err* file is opened, the condition checking of step one through three will be repeated. However, the action will store the value of first_mess, mess_number, second_mess, or all of them in the *Graptool.err* file.

```
/* CHECK AN ERROR FILE IS OPENED */
if(error_file_open)
 {
/* CHECK FOR AN ENDARRAY CHARACTER */
 if(*first_mess != ENDARRAY)
   fprintf(eptr, "%s", first_mess);
/* CHECK FOR A NEGATIVE ONE */
 if(mess_number != NONE)
  fprintf(eptr, "%d", mess_number);
/* CHECK FOR AN ENDARRAY CHARACTER */
 if(*second_mess != ENDARRAY)
  fprintf(eptr, "%s", second_mess);
 }
```

Step 5.: The **Message** will return to the calling function.

```
}
```



Purpose: An Error function will display the error message and stop Graptool software execution.

Prototype:

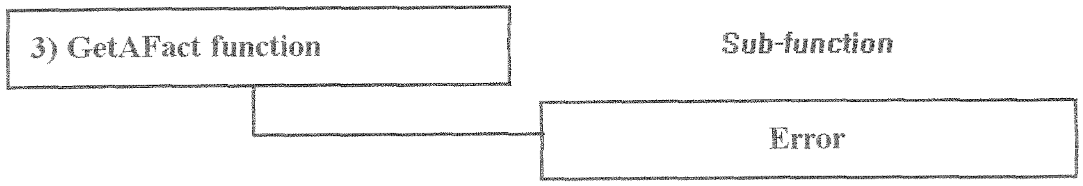
```
void Error(char *first_errmess, int err_number, char *second_errmess)
{
```

A first_errmess and a second_errmess are character pointer variables which contain an address of an error message array.
An err_number is an integer number variable.

Pseudo-code:

An Error function will send the address of a first_errmess, the value of an err_number, and the address of a second_errmess to the Message function. After that, the computer will close all the opened files and return control back to the DOS operating system.

```
/* DISPLAY THE ERROR MESSAGE ON A COMPUTER SCREEN */
Message("\n\n ERROR", NONE, first_errmess);
Message(ENDARRAY, err_number, second_errmess);
/* CLOSE ALL THE OPENED FILES */
fcloseall();
/* RETURN THE CONTROL TO DOS OPERATING SYSTEM */
exit(1);
}
```



Purpose: A <u>**GetAFact**</u> function will get a **block** from a given array.

Prototype:

```
char *GetAFact(char *array_ptr, int fact_type, char a_fact[])
{
```

An **array_ptr** is a character pointer variable that contains an ending array address of the previous **block** which has already been read.

A **fact_type** is an integer variable which contains the ASCII number of a DEFFACTS, a CONDITION, an ASSERT, or a RETRACT character. A DEFFACTS, a CONDITION, an ASSERT, and a RETRACT character indicate the function to read a **fact block**, a **condition block**, a **new block**, and a **removing block**, respectively.

An **a_fact** is an array character which passes an already being read **block** to the calling function.

Pseudo-code:

Step 1.: A computer first assigns an O_PARENTHES character to the ***facttype*** variable (identify the **block**) and ENDARRAY character to the ***endch*** variable (indicate the end of the array). After that, the computer will check a value inside the **fact_type**. If a value is not a DEFFACTS character, the computer will assign ENDRF character to the ***endch*** and a value of the **fact_type** to the ***facttype***.

```
char facttype = O_PARENTHES, endch = ENDARRAY;
.* CHECK IT IS NOT AN DEFFACTS */
if(fact_type != DEFFACTS)
 {
  endch = (char)ENDRF;
  facttype = (char)fact_type;
 }
```

Step 2.: The computer will start the searching of the ***facttype*** value from a given array address (in **array_ptr**) until it finds that value or reaches the end of an array (***endch*** value). If the ***facttype*** value is found, the computer will start to copy it character by character after that value until the computer copy a closed parenthesis (the end of a **block**). The result will be stored in the **a_fact** array.

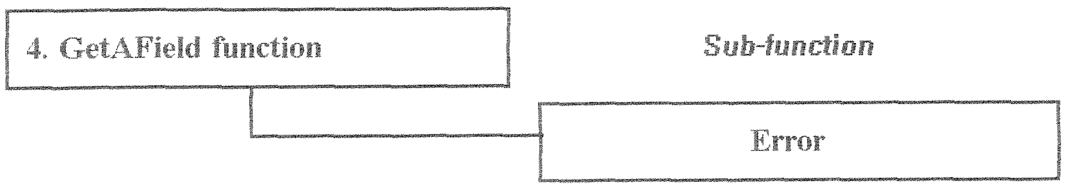
```
/* SEARCH FOR A facttype CHARACTER IN THE GIVEN ARRAY */
while((*array_ptr != facttype) && (*array_ptr != endch))
   array_ptr++;
/*CHECK FOR THE EXISTENCE OF A facttype CHARACTER */
if(*array_ptr == facttype)
 {
 if(*array_ptr == O_PARENTHES)
   array_ptr--;
 /* READ A FACT FROM THE GIVEN ARRAY TO AN a_fact ARRAY */
 do
  {
   array_ptr++;
   a_fact[i] = *array_ptr;
   +;
  }
 while((*array_ptr != C_PARENTHES) && (i < FACT_SIZE-2));
 a_fact[i] = ENDARRAY;
}
```

Step 3.: If the size of **a_fact** is in excess, the computer will call the **Error** function to display an error message..

```
/*CHECK FOR THE SIZE OF RETURNING FACT */
if(i >= FACT_SIZE-2)
 Error("FIND A FACT TOO LONG FOR THE FACT ARRAY.", -1, "\n");
```

Step 4.: The **GetAFact** will return to the calling function with the **array_ptr** value (ending address of an given array which equal to ***endch*** value or ending address of being read **block** which is a closed parenthesis).

```
return(array_ptr);
}
```



Purpose: A <u>**GetAField**</u> function will get a **field** from a **block**.

Prototype:

```
char *GetAField(char *fact_ptr, char a_field[], int *notlast)
{
```

A fact_ptr is a character pointer variable that contains an ending address of the previous **field** which has already been read.

An a_field is a character array which passes an already being read **field** to the calling function.

A notlast is an integer variable. The notlast is TRUE (one) if the **field** being read is the **last field** in the **block**. Otherwise, the notlast is FALSE (zero).

Pseudo-code:

Step 1.: The computer will first assign a FALSE to the notlast. After that, the computer will start to copy from the previous **field** ending address (in fact_ptr), until it reaches a space (the end of a **field**) or closed parenthesis (the end of a **block**). The result will be stored in the a_field array.

```
*notlast = FALSE;
/* READ A FILED FROM THE GIVEN FACT TO AN a_field ARRAY */
while((*fact_ptr != SPACE)&&(*fact_ptr != C_PARENTHES)&&(i < FIELD_SIZE-2))
 {
  a_field[i] = *fact_ptr;
  fact_ptr++;
  +;
 }
a_field[i] = ENDARRAY;
```

Step 2.: If the size of the a_field array is in excess, the computer will send an error message to the <u>**Error**</u> function.

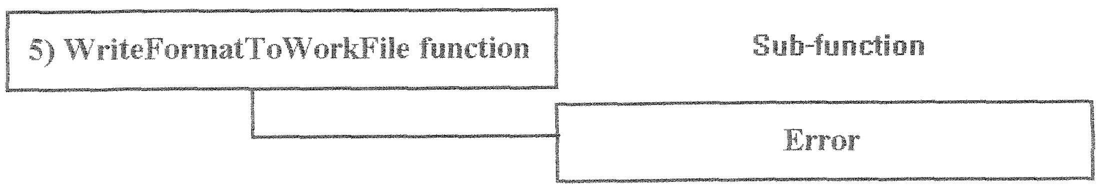
```
/* CHECK FOR THE RETURNING FIELD SIZE */
if(i >= FIELD_SIZE-2)
 Error("FIND A FIELD TOO LARGE FOR THE FIELD ARRAY.", -1, "\n");
```

Step 3.: The computer will check for the end of a given **block**. If the end of a **block** (closed parenthesis) has not been reached, **notlast** will be set to TRUE. Otherwise, it will remain FALSE.

```
/* CHECK IT IS NOT THE LAST FIELD IN A FACT */
if(*fact_ptr != C_PARENTHES)
  *notlast = TRUE;
```

Step 4.: The **GetAField** will return to the calling function with the **fact_ptr** value (ending address of a **block** which is a closed parenthesis or ending address of being read **field** which is a space).

```
return(fact_ptr);
}
```



Purpose: A <u>**WriteFormatToWorkingFile**</u> function will write a **block** into a given working file. This function also checks the initialization of a ***fact_base*** array.

Prototypes:

```
char *WriteFormatToWorkingFile(int fact_type, char *fact_ptr, FILE *workfile)
{
```

A **fact_type** is an integer number variable which contains the ASCII number of a DEFFACTS, a DEFRULE, a CONDITION, an ASSERT, or a RETRACT character. This character will be written in the front of a **block** that is being written.
A **fact_ptr** is a character pointer variable that contains an address of a **block** which will be written to the working file.
A **workfile** is a file pointer variable which contains an address of a working file.

Pseudo-code:

Step 1.: The computer will first write a **fact_type** to the selected working file. After that, the computer will start to write character by character from a given address (in the **fact_ptr**) to the first closed parenthesis (the end of a **block**) into the working file.

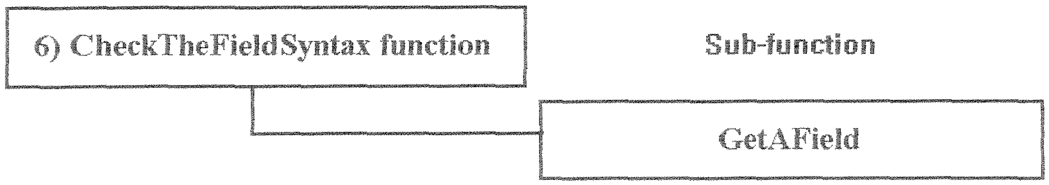
```
/* WRITE THE GRAPTOOL FORMAT BLOCK TO THE WORKING FILE */
putc(fact_type, workfile);
while(*fact_ptr != C_PARENTHES)
 {
  putc(*fact_ptr, workfile);
  block_size++;
  fact_ptr++;
 }
putc(C_PARENTHES, workfile);
```

Step 2.: The computer will do an approximate calculation of the fact base size and compares it to the size of the ***fact_base*** array. If the ***fact_base*** array is too small, the computer will call an **Error** function to display an error message.

```
/* CALCULATE AN APPROXIMATION SIZE OF FACT BASE */
if(fact_type == RETRACT)
  fact_base_size = fact_base_size - block_size;
else
if((fact_type == ASSERT) || (fact_type == DEFFACTS))
  fact_base_size = fact_base_size + block_size;
/* CHECK THE SIZE OF fact_base ARRAY */
if(fact_base_size >= CON_SIZE-2)
  Error("CON_SIZE, ",CON_SIZE,", BYTES IS TOO SMALL. \n");
```

Step 3.: The **WriteFormatToWorkingFile** will return to the calling function with the **fact_ptr** value which is an ending address of a **block** (a closed parenthesis).

```
return(fact_ptr);
}
```



Purpose: A <u>**CheckTheFieldSyntax**</u> function will check the syntax errors of each **field** in the given **fact block**.

Prototype:

```
int CheckTheFieldSyntax(char *fact_ptr)
{
```

A **fact_ptr** is a character pointer variable which contains an address of a **fact block**.

Pseudo-code:

Step 1.: The computer calls the <u>**GetAField**</u> function to get a **field** from a given **fact block** (an address is in the **fact_ptr**). The **field** being read will be stored in an ***a_field*** array.

```
/* CHECK THE SYNTAX OF EACH FIELD IN A FACT BLOCK */
do
 {
 /* GET A FIELD FROM THE GIVEN FACT BLOCK */
 fact_ptr = GetAField(fact_ptr, a_field, &notlast);
```

Step 2.: If the ***a_field*** is not a **string**, the computer will do the sub-step two. Otherwise, the computer will go to step three.

```
 /* CHECK IT IS NOT A STRING (BEGIN AND END WITH QUOTE) */
 if(a_field[0] != QUOTE)
  {
```

Step 2.1.: If the ***a_field*** is a **wildcard** or **variable**, the ***good_fact*** variable will be set to FALSE and go to step three. Otherwise the **good_fact** remains TRUE.

```
  /* CHECK IT IS A WILDCARD OR VARIABLE FIELD */
  if((a_field[0] == S_WILDCARD) ||
        ((a_field[0] == M_WILDCARD) && (a_field[1] == S_WILDCARD)))
   good_fact = FALSE;
```

Step 2.2.: If the *good_fact* is still TRUE, the computer will check for the existence of any logical operators and parentheses in the ***a_field*** array. The *good_fact* will be set to FALSE if the computer finds any logical operators or parentheses.

```
/* CHECK good_fact VALUE IS TRUE */
if(good_fact)
 {
 /* CALCULATE THE SIZE OF a_field ARRAY */
 length = strlen(a_field);
 /* CHECK FOR THE EXISTENCE OF ANY LOGICAL */
 /* OPERATORS OR PARENTHESES IN THE FIELD */
 for(i=0; i< length; i++)
   {
    if(((a_field[i] == LOGI_OR) || (a_field[i] == LOGI_AND) ||
                 (a_field[i] == ':')) && (a_field[i] != a_field[i+1]))
       good_fact = FALSE;
    else
    if((a_field[i] == LOGI_NOT) ||
                 (a_field[i] == O_PARENTHES) || (a_field[i] == C_PARENTHES))
      good_fact = FALSE;
   }
 }
}
```

Step 3.: The computer will go back to step one if the **field** (in ***a_field***) is not the **last field** (*notlast* variable is TRUE) in the given **fact block**.

```
 }
while(notlast);
```

Step 4.: The **<u>CheckTheFieldSyntax</u>** will return to the calling function with the *good_fact* value. The *good_fact* value is TRUE for a good **fact block** and FALSE for the existence of any error.

```
return(good_fact);
}
```

7) LogicalOperation function

Purpose: A <u>**LogicalOperation**</u> function will compare a given **field** from a **fact block** of the ***fact_base*** array with a given logical field from a **condition block** of the ***rule_base*** array.

Prototype:

```
int LogicalOperation(char ffield[], char cfield[])
{
```

A ffield is a character array which contains a logical field from a **condition block** of the ***rule_base*** array.
A cfield is a character array which contains a **field** from a **fact block** of the ***fact_base*** array.

Pseudo-code:

Step 1.: The computer will check the *while loop* condition which is the ffield[fptr] value. If ffield[fptr] does not equal ENDARRAY character (the computer does not reach the end of the logical field), the computer will do step two. Otherwise, the computer go to step nine.

```
/* DO A FIELD  LOGICAL OPERATION */
while(ffield[fptr] != ENDARRAY)
 {
```

Step 2.: Because a given logical field (in the ffield) can be the combination of **fields**, the computer will copy a sub-logical field (separated by a logical **and** or logical **or**) from the ffield into a ***subfield*** array.

```
 sptr = 0;
 /* READ A SUB-LOGICAL FIELD FROM ffield TO subfield ARRAY*/
 do
  {
   subfield[sptr] = ffield[fptr];
   sptr++;
   fptr++;
  }
 while((ffield[fptr] != LOGI_AND) &&
          (ffield[fptr] != LOGI_OR) && (ffield[fptr] != ENDARRAY));
 subfield[sptr] = ENDARRAY;
```

Step 3.: The computer will first set a *current* variable to FALSE. If the ***subfield*** has a logical **and** or logical **or**, the computer will copy that logical operator to the *logical* variable.

```
tptr = current = FALSE;
/* GET A LOGICAL OPERATOR FROM A subfield */
if((subfield[0] == LOGI_AND) || (subfield[0] == LOGI_OR))
 {
  logical = subfield[0];
  tptr++;
 }
```

Step 4.: If the ***subfield*** has a logical **not**, that logical **not** will be hidden before the computer compares the ***subfield*** with the cfield. If the ***subfield*** array matches the cfield, the *current* will be set to TRUE. Otherwise, the *current* remains FALSE.

```
/* CHECK FOR AN EXISTENCE OF LOGICAL NOT IN A subfield */
if(subfield[tptr] == LOGI_NOT)
 tptr++;
/* COMPARE subfield WITH cfield ARRAY*/
if(strcmp(&subfield[tptr], cfield) == SUCCESS)
 current = TRUE;
```

Step 5.: If the ***subfield*** has a logical operator **not**, the *current* value will be set to the opposite value (TRUE become FALSE and vice versa).

```
/* INVERSE A current VALUE IF subfield HAS A LOGICAL NOT */
if((tptr != 0) && (subfield[tptr-1] == LOGI_NOT))
 current = !current;
```

Step 6.: If it is the first time the computer performs the logical operation, the computer will take the *current* value as the logical result (assign the *current* value to a *match* variable) and go to step eight.

```
/* CHECK IT IS THE FIRST LOGICAL OPERATION */
if(previous == NONE)
 match = current;
```

Step 7.: If it is not the first time the computer performs the logical operation, the computer will do the logical operation with the previous field (its value is in the *previous* variable). The result of this step will be saved in the *match* variable.

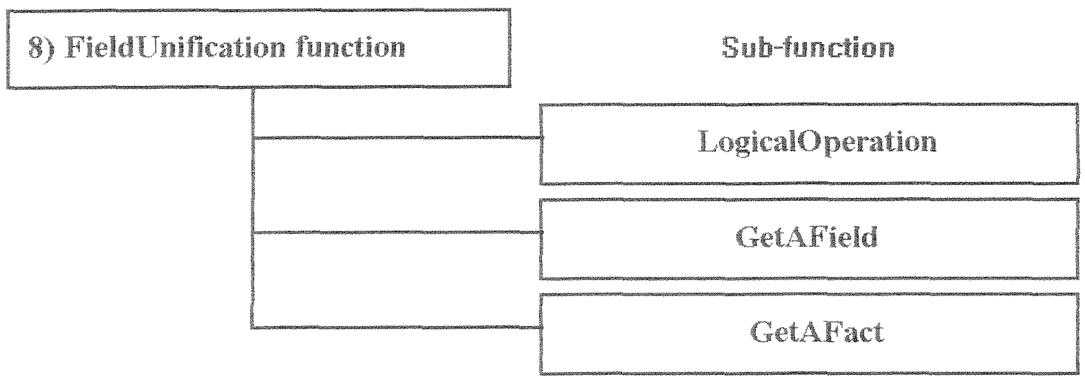
```
else
/* DO THE LOGICAL OPERATION WITH PREVIOUS FIELD */
switch(logical)
 {
  case LOGI_AND :if(current && previous)
                  match = TRUE;
                else
                  match = FALSE;
                break;
  case LOGI_OR :if(current || previous)
                  match = TRUE;
                else
                  match = FALSE;
                break;
  default     :match = current;
 }
```

Step 8.: The computer assigns a *current* value to the *previous* variable and goes back to step one.

```
/* ASSIGN A current VALUE TO A previous VARIABLE */
previous = current;
}
```

Step 9.: The **LogicalOperation** will return to the **FieldUnification** function with the result of the logical operation (in *match*), which can be either TRUE or FALSE.

```
return(match);
}
```



Purpose: A FieldUnification function will check for any matching between two given **blocks**. One of the **blocks** comes from the ***rule_base*** array and the other **block** comes from the ***fact_base*** array. This function will compare two **fields** from each **block** at a time. If the **field** from the ***rule_base*** array block is a **variable field**, the function will store that field and its values in the ***variable*** array.

Prototype:

```
int FieldUnification(char fact_str[], char con_str[])
{
```

A fact_str is a character array which contains a **block** from the ***rule_base*** array. This **block** will be either a **new block**, a **removing block**, or a **condition block**.

A con_str is a character array which contains a **block** from the ***fact_base*** array. This **block** is a **fact block**.

Pseudo-code:

Step 1.: The computer assigns the address of fact_str and con_str to *fptr* and *cptr* which are character pointer variables. The computer also assigns a zero to the variables *fcount* and *ccount*.

```
/* ASSIGN THE fact_str AND con_str ADDRESSES TO fptr AND cptr VARIABLE*/
fptr = fact_str;
cptr = con_str;
fcount = ccount = 0;
```

Step 2.: The computer calculates the number of **fields** in fact_str and con_str. The number of **fields** in fact_str is stored in the variable named *ftotal*. The number of **fields** in *con_str* is stored in *ctotal*. The computer will then re-initialize the value of *fptr*, *cptr*, *fcount*, and *ccount*, as in step one.

```
/* CALCULATE THE TOTAL NUMBER OF FIELDS IN THE fact_str */
do
 {
  fcount++;
  fptr = GetAField(fptr, ffield, &fnotlast);
 }
while(fnotlast);
/* CALCULATE THE TOTAL NUMBER OF FIELDS IN THE con_str */
do
 {
  ccount++;
  cptr = GetAField(cptr, cfield, &cnotlast);
 }
while(cnotlast);
/* ASSIGN NUMBER OF FIELDS TO ftotal AND ctotal */
ftotal = fcount;
ctotal = ccount;
/* RE-INITIALIZE THE VALUE OF  fptr, cptr, fcount, AND ccount */
fptr = fact_str;
cptr = con_str;
fcount = ccount = 0;
```

Step 3.: The computer will assign the address in *fptr* and *cptr* to the pointer variables named *f_ptr* and *c_ptr* because the computer will need this address later on in the process. The computer will also assign FALSE (zero) to *match* and *logical*.

```
/* DO THE FIELD UNIFICATION PROCESS */
do
 {
  f_ptr = fptr;
  c_ptr = cptr;
  match = logical = FALSE;
```

Step 4.: The computer will get a **field** from fact_str and con_str. A **field** from fact_str is stored in the ***ffield*** array and its address is stored in *fptr*. A **field** from con_str is stored in the ***cfield*** array and its address is stored in *cptr*. The computer also updates the field number of each **field** in increments of the value of *fcount* (contains the field number of ***ffield***) and *ccount* (contains the field number of ***cfield***) by one.

```
 /* GET A FIELD FROM fact_str  AND STORE IT IN ffield ARRAY */
 fptr = GetAField(fptr, ffield, &fnotlast);
 /* GET A FIELD FROM con_str AND STORE IT IN cfield ARRAY */
 cptr = GetAField(cptr, cfield, &cnotlast);
 /* UPDATE THE FIELD NUMBER OF ffield AND cfield */
 fcount++;
 ccount++;
```

Step 5.: The computer will check if ***ffield*** is or contains a logical field. The ***ffield*** will be considered a logical field if it contains any logical characters (LOGI_NOT "~", a LOGI_AND "$", or a LOGI_OR "|") and does not begin with a QUOTE character. If it is true that the ***ffield*** is/has the logical field, the computer will do the following:

The computer will set the ***logical*** variable to TRUE and assign the address of ***ffield*** to the ***ptr*** pointer variable.

If the ***ffield*** is a **variable field** that contains any sub-logical field, the computer will search the beginning address of that sub-field and assign it to the ***ptr*** variable.

The computer will call the **LogicalOperation** function to do the logical operation between the logical field of ***ffield*** (the address is inside ***ptr***) and ***cfield***. The result of this operation will be stored in the ***match*** variable.

If the address of ***ffield*** is not the ***ptr*** value, the computer will assign the ENDARRAY character to the address inside ***ptr*** to erase the sub-logical field of ***ffield***.

The computer will go to step seven.

```
/* CHECK FOR THE EXISTENCE OF ANY LOGICAL OPERATORS INSIDE ffield */
if(((strchr(ffield, LOGI_NOT) != ENDARRAY) ||
        (strchr(ffield, LOGI_AND) != ENDARRAY) ||
                (strchr(ffield, LOGI_OR) != ENDARRAY)) &&
                        (ffield[0] != QUOTE))
 {
 logical = TRUE;
 ptr = ffield;
 /* CHECK IF ffield IS A VARIABLE FIELD */
 if(ffield[0] == S_WILDCARD)
   /* SEARCH FOR THE BEGINNING ADDRESS OF LOGICAL FIELD IN ffield */
   ptr = strchr(ffield, LOGI_AND);
 /* DO THE LOGICAL OPERATION BETWEEN ffield AND cfield */
 match = LogicalOperation(++ptr, cfield);
 /* CHECK IF THE ADDRESS INSIDE ptr IS NOT AN ffield ADDRESS */
 if(ptr != ffield)
   /* ERASE THE SUB-LOGICAL FIELD INSIDE ffield */
   *(--ptr) = ENDARRAY;
 }
```

Step 6.: If the ***ffield*** does not have any logical fields, the computer will compare ***ffield*** with a ***cfield***. If they do not match, both fields will be converted into real numbers before another comparison is done. If ***ffield*** matches ***cfield*** (before or after the conversion), the *match* variable will be set to TRUE. Otherwise, the *match* remains FALSE.

```
else
/* COMPARING ffield WITH cfield */
if(strcmp(ffield, cfield) == SUCCESS)
  match = TRUE;
else
 {
  /* CONVERT ffield AND cfield TO REAL NUMBERS */
  fvalue = strtod(ffield, &fendptr);
  cvalue = strtod(cfield, &cendptr);
   /* COMPARING THE REAL NUMBERS OF ffield AND cfield */
  if((fvalue == cvalue)&&(*fendptr == ENDARRAY)&&(*cendptr == ENDARRAY))
    match = TRUE;
 }
```

Step 7.: The computer will check if ***ffield*** is a **wildcard** or a **variable field**. The ***ffield*** will be considered a **wildcard** or a **variable field** if it begins with an S_WILDCARD (?) or the combination of M_WILDCARD with S_WILDCARD ($?). If it is true that ***ffield*** is a **wildcard** or a **variable field**, the computer will do sub-step 7. Otherwise, it will skip to step eight.

```
/* CHECK IF ffield IS A WILDCARD OR VARIABLE FIELD */
if((ffield[0] == S_WILDCARD)
          || ((ffield[0] == M_WILDCARD) && (ffield[1] ==  S_WILDCARD)))
 {
```

Step 7.1.: The computer will first check the value of *logical*. If *logical* is FALSE (***ffield*** dose not have any logical fields.), the computer will set *match* to TRUE. Otherwise the *match* value remains unchanged. The computer also sets the *found* variable to FALSE, assigns an address of ***variable*** array to the *vptr* pointer variable, and sets the *var_match* variable to TRUE.

```
  /* CHECK IF ffield IS OR HAS ANY LOGICAL FIELDS */
  if(!logical)
    match = TRUE;
   /* INITIALIZE THE VALUE OF found, vptr, AND var_match */
  found = FALSE;
  vptr = variable;
  var_match = TRUE;
```

Step 7.2.: The computer will check for the existence of ***ffield*** in the ***variable*** array. If ***ffield*** is found, the computer will set *found* to TRUE and search for the ***ffield*** values from the ***variable*** array inside con_str. This search will start from ***cfield*** through the last field of con_str. If any of the ***ffield*** values from the ***variable*** array does not exist in con_str, the computer will set *var_match* to FALSE.

```
/* CHECK FOR THE EXISTENCE OF ffield IN THE variable ARRAY */
while((!found) && (*(vptr = GetAFact(vptr,
                              DEFFACTS, var_str)) != (char)ENDARRAY))
 {
 vfptr = var_str;
  /* GET A FIELD FROM variable ARRAY */
 vfptr = GetAField(vfptr, vfield, &vnotlast);
 /* CHECK IF ffield EXISTS IN THE variable ARRAY */
 if(strcmp(ffield, vfield) == SUCCESS)
   {
    cptr = c_ptr;
    found = TRUE;
    /* CHECK FOR THE EXISTENCE OF ffield VALUES */
     /* FROM variable ARRAY INSIDE THE con_str */
    do
      {
       /* GET A FIELD FROM con_str */
       cptr = GetAField(cptr, cfield, &cnotlast);
        /* GET A FIELD FROM variable ARRAY */
       vfptr = GetAField(vfptr, vfield, &vnotlast);
        /* CHECK IF ffield VALUE EXISTS INSIDE con_str */
        if(strcmp(vfield, cfield) != SUCCESS)
          var_match = FALSE;
      }
    while((var_match) && (vnotlast));
   }
 }
```

Step 7.3.: The ***ffield*** matches the ***cfield*** if and only if both *match* and *var_match* are TRUE. The computer will set *match* to TRUE if *match* and *var_match* are TRUE. Otherwise, the computer will set *match* to FALSE.

```
/* THE ffield MATCHES THE cfield */
/*IF AND ONLY IF  BOTH match AND var_match ARE TRUE */
if(match && var_match)
 match = TRUE;
else
 match = FALSE;
```

Step 7.4.: The computer will check the value of *found* variable. If the *found* value is FALSE, the computer will update the ***variable*** array. This process is started by setting the *notvar* variable to FALSE, setting *i* to the current ending of ***variable*** array, and then adding an open parenthesis and ***ffield*** to the end of ***variable*** array. After that, the computer will do sub-step 7.4. However, if the *found* variable is TRUE, the computer will skip to step eight.

```
/* CHECK FOR THE EXISTENCE  OF ffield IN variable ARRAY*/
if(!found)
 {
 notvar = FALSE;
 /* SET i TO CURRENT ENDING OF variable ARRAY */
 i = strlen(variable);
 /* ADD AN OPEN PARENTHESIS AND ffield TO variable ARRAY */
 strcat(variable, "(");
 strcat(variable, ffield);
```

Step 7.4.1. The computer will first check if ***ffield*** is a **single-field wildcard** or a **single-field variable** by checking if the first character in ***ffield*** is a S_WILDCARD (?) character. After that, the computer will separate a **single-field wildcard** from a **single-field variable** by checking the second character in ***ffield***. If the second character of ***ffield*** is an ENDARRAY (null) character, the ***ffield*** is a **single-field wildcard**. The computer will then set *notvar* to TRUE. In this step, the computer also sets the *endread* value to the ccount value (the field number of ***ffield***) plus one. The *endread* will be used later on to indicate the ending of the ***ffield*** value in the con_str.

```
/* CHECK IF ffield IS A SINGLE-FIELD VARIABLE OR WILDCARD */
if(ffield[0] == S_WILDCARD)
 {
 /*CHECK IF ffield IS A SINGLE-FIELD WILDCARD */
 if(ffield[1] == ENDARRAY)
   notvar = TRUE;
 endread = ccount + 1;
 }
```

Step 7.4.2. The computer will first check if ***ffield*** is a **multifield wildcard** or a **multifield variable** by checking if the first two characters in the ***ffield*** are a M_WILDCARD ($) character and a S_WILDCARD (?) characters. After that, the computer will separate a **multifield wildcard** from a **multifield variable** by checking the third character in the ***ffield***. If the third character of ***ffield*** is an ENDARRAY character, the ***ffield*** is a **multifield wildcard**. The computer will then set *notvar* to TRUE. In this step, the value of *endread* depends on the location of ***ffield*** in the fact_str which are divided into three categories:

```
/* CHECK IF ffield IS A MULTIFIELD VARIABLE OR WILDCARD */
if((ffield[0] == M_WILDCARD) && (ffield[1] == S_WILDCARD))
  {
  /* CHECK IF ffield IS A MULTIFIELD WILDCARD */
  if(ffield[2] == ENDARRAY)
   notvar = TRUE;
```

If ***ffield*** is the **last field** in **fact_str** (*fnotlast* is FALSE), *endread* will be the value of *ctotal* (the field number of **last field** in **con_str**) plus one and the computer will go to step 7.4.3.

```
/* CHECK IF ffield IS THE LAST FILED IN fact_str */
if(!fnotlast)
 /* UPDATE THE endread VALUE */
 endread = ctotal + 1;
```

If ***ffield*** is not the **last field** in **fact_str**, the computer will search for the boundary of ***ffield*** by getting a field after the ***ffield*** in **fact_str**. If a field after ***ffield*** is a **wildcard** or a **variable field**, the *endch* value will be the field number of a **field** in the **con_str** (which has the same backward position with a field after the ***ffield*** from the **fact_str**). However, if the field after ***ffield*** is not a **wildcard** or a **variable field**, the computer will first set the *endch* value to the *ccount* value. It will then search for the matching **field** of a field after ***ffield*** in the **con_str**. The *endread* value will be the field number of that matching field and the computer will go to step 7.4.3.

```
else
 f_ptr = fptr;
 /* GET THE BOUNDARY FIELD OF ffield */
 fptr = GetAField(fptr, ffield, &fnotlast);
 fptr = f_ptr;
 fnotlast = TRUE;
 /* CHECK IF BOUNDARY FIELD IS A VARIABLE FIELD OR WILDCARD */
 if((ffield[0] == S_WILDCARD) ||
         ((ffield[0] == M_WILDCARD) && (ffield[1] == S_WILDCARD)))
  /* UPDATE THE endread VALUE */
  endread = ctotal - (ftotal - fcount + 1) + ccount;
 else
  {
   /* INITIALIZE endread AND cptr */
   endread = ccount;
   cptr = c_ptr;
   /* SEARCH FOR THE BOUNDARY FIELD IN THE con_str */
   do
     {
     cptr = GetAField(cptr, cfield, &cnotlast);
     if(strcmp(ffield, cfield) != SUCCESS)
      /* UPDATE THE endread VALUE */
      endread++;
     else
      cptr = con_str;
     }
   while((cnotlast) && (cptr != con_str));
  }
```

If the computer cannot find the value of ***ffield*** in the **con_str** for any reason (the *endread* value equals the *ccount* value), the computer will set the *cnotlast* and the *notvar* to TRUE and go to step 7.4.3.

```
        /*CHECK IF THE ffield VARIABLE IS BOUND TO NOTHING */
        if(endread == ccount)
          cnotlast = notvar = TRUE;
      }
    }
```

Step 7.4.3.: The computer will write the values of ***ffield*** from **con_str** into the ***variable*** array by starting from ***cfield*** until *endread* field is reached. If ***ffield*** is not a **variable field** (*notvar* is TRUE), the computer will reset the ending of the ***variable*** array to the previous ending (*i* value).

```
    cptr = c_ptr;
     /* ADD ffield VALUE TO THE variable ARRAY */
    for(j = ccount; j<endread; j++)
      {
       cptr = GetAField(cptr, cfield, &cnotlast);
       strcat(variable, " ");
       strcat(variable, cfield);
      }
    strcat(variable, ")");
    /* CHECK IF ffield IS NOT A VARIABLE FIELD */
    if(notvar)
      /* RE-SETTING THE END OF variable ARRAY */
      variable[i] = ENDARRAY;
    }
   }
```

Step 8.: The computer will go back to do step three if it has not reached the end of **fact_str** (*fnotlast* is TRUE), the end of **con_str** (*cnotlast* is TRUE), and if ***ffield*** matches ***cfield*** (*match* is TRUE).

```
  }
while((match) && (fnotlast) && (cnotlast));
```

Step 9.: If the **last field** of **fact_str** has not been checked (*fnotlast* does not equal to SUCCESS) and *match* is TRUE, the computer will get that **last field**. If that field is not a **multifield wildcard** or **multifield variable**, the *match* will be set to FALSE.

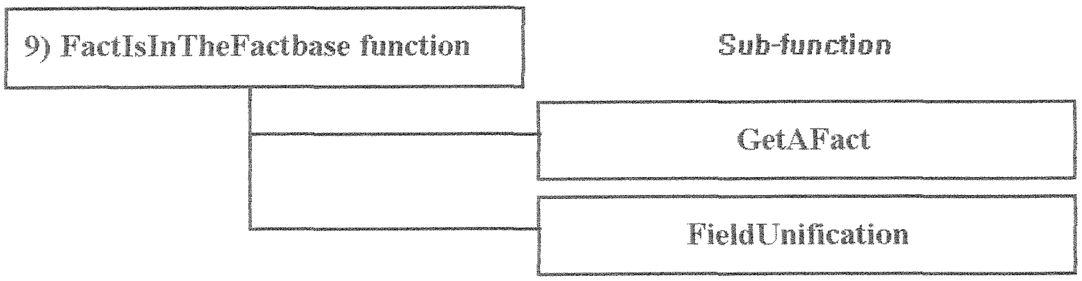
```
/* CHECK fact_str IF LAST FIELD HAS NOT BEEN CHECKED */
if((fnotlast != SUCCESS) && (match))
 {
 GetAField(fptr, ffield, &fnotlast);
 /* CHECK fact_str  IF LAST FIELD IS NOT A */
 /*MULTIFIELD WILDCARD  OR VARIABLE */
 if((ffield[0] != M_WILDCARD) || (ffield[1] != S_WILDCARD))
   match = FALSE;
 }
```

Step 10.: If fact_str or con_str have not reached the end, the *match* variable will be set to FALSE.

```
/* CHECK IF fact_str OR con_str HAS NOT REACHED THE END */
if((fnotlast != SUCCESS) || (cnotlast != SUCCESS))
  match = FALSE;
```

Step 11.: The **FieldUnification** returns to the **FactIsInTheFactbase** function with the *match* value (TRUE or FALSE).

```
return(match);
}
```



Purpose: A <u>**FactIsInTheFactbase**</u> function will check for the existence of a given **block** from the ***rule_base*** array in the ***fact_base*** array.

Prototype:

```
char *FactIsInTheFactbase(char fact_str[], int *found)
{
```

A fact_str is a character array which contains a **block**. from the ***rule_base*** array.

A found is an integer variable which contains the existence status of fact_str in the ***fact_base*** array.

Pseudo-code:

Step 1.: While the fact_str does not exist in the ***fact_base*** array (*match* variable is FALSE) and the end of ***fact_base*** array (an ENDARRAY character) has not been reached, the computer will assign address in *c_ptr* to the *match_ptr* and then get a **fact block** from the ***fact_base*** array by the <u>**GetAFact**</u> function. The **fact block** being read is stored in an ***a_fact*** array. The <u>**FieldUnification**</u> function will compare the fact_str with ***a_fact*** array. The result of this comparison will be stored in the *match* variable. If the *match* value is FALSE, the computer will reset the end of ***variable*** array.

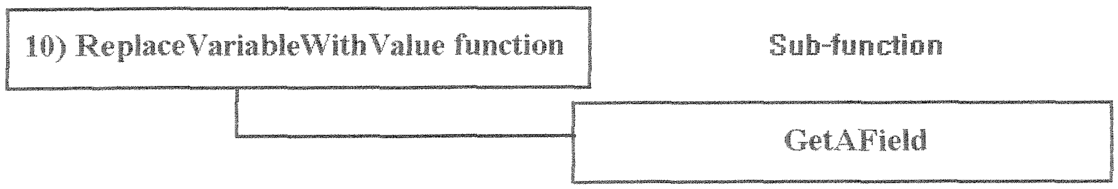
```
c_ptr = fact_base;
endvar = strlen(variable);
/* DO THE SEARCHING OF A GIVEN FACT IN A fact_base ARRAY */
while((!match) && (*c_ptr != (char)ENDARRAY))
 {
  match_ptr = c_ptr;
 /* COMPARE A GIVEN FACT WITH A FACT FROM THE fact_base ARRAY */
 if(*(c_ptr = GetAFact(c_ptr, DEFFACTS, a_fact)) != (char)ENDARRAY)
   match = FieldUnification(fact_str, a_fact);
 /* CHECK match IS NOT TRUE */
 if(!match)
   variable[endvar] = ENDARRAY;
  c_ptr++;
 }
```

Step 2.: The result of step one which is the ***match*** value, will be assigned to the *found.

```
/* ASSIGN A match VALUE TO A found VARIABLE */
*found = match;
```

Step 3.: The **FactIsInTheFactbase** will return to the calling function with the ***match_ptr*** value (address of the matching **fact block** in the ***fact_base*** array, or the ending address of the ***fact_base*** array).

```
return(match_ptr);
}
```



Purpose: A <u>ReplaceVariableWithValue</u> function will replace all the variable fields in the given **action block** with its values from the ***variable*** array.

Prototype:

```
void ReplaceVariableWithValue(char fact_str[], char result_str[])
{
```

A fact_str is a character array which contains the **action block**.

A result_str is a character array which contains the result of the function processes.

Pseudo-code:

Step 1.: The computer will assign the fact_str value to the *fact_ptr* variable and copy an open parenthesis to the beginning of result_str.

```
/* INITIALIZE THE VARIABLE */
fact_ptr = fact_str;
stripy(result_str, "(");
```

Step 2.: While the **last field** of fact_str array has not been read (*fnotlast* is TRUE), the computer will do step three. Otherwise, the computer will go to step four.

```
/* DO THE VALUE REPLACEMENT PROCESSES */
while(fnotlast)
```

Step 3.: The computer will call the <u>GetAField</u> function to get a **field** from the fact_str array. The **field** being read is saved in the ***ffield*** array. If the ***ffield*** is a variable field, the computer will do step 3.1, 3.2 and then go back to step two.. Otherwise, the computer will do step 3.3 and then go back to step two..

```
 {
 /* GET A FIELD FROM THE fact_str ARRAY */
 fact_ptr = GetAField(fact_ptr, ffield, &fnotlast);
 /* CHECK ffield IS A VARIABLE FIELD */
 if((ffield[0] == S_WILDCARD) ||
              ((ffield[0] == M_WILDCARD) && (ffield[1] == S_WILDCARD)))
  {
```

Step 3.1.: If the ***ffield*** contains any logical fields, the computer will assign an ENDARRAY character to the beginning of that logical field.

```
  /* CHECK ffield CONTAINS ANY LOGICAL FIELDS */
  if((ffptr = strchr(ffield, LOGI_AND)) != ENDARRAY)
   *ffptr = ENDARRAY;
```

Step 3.2.: The computer will search for the ***ffield*** values in the ***variable*** array. If the ***ffield*** values have been found, the computer will write it into the **result_str** array.

```
  vptr = variable;
  notdone = TRUE;
  /*SEARCH FOR THE VALUES OF ffield IN A variable ARRAY */
  while((notdone) && (*vptr != ENDARRAY))
   {
   vptr = GetAField(vptr, vfield, &vnotlast);
   if(strcmp(ffield, vfield) == SUCCESS)
    {
     /* ADD VALUES FROM A variable ARRAY TO THE result_str ARRAY */
     while(vnotlast)
      {
      vptr = GetAField(vptr, vfield, &vnotlast);
      strcat(result_str, vfield);
      strcat(result_str, " ");
      }
     notdone = FALSE;
    }
   /* SEARCH FOR THE BEGINNING OF VARIABLE FIELD IN A variable ARRAY */
   vptr = strchr(vptr, O_PARENTHES);
   }
  }
```

Step 3.3.: If the ***ffield*** array is not the variable field, the computer will add it to the **result_str** array.

```
 else
  /* IF ffield IS NOT A VARIABLE FIELD, ADD IT TO THE result_str ARRAY */
  {
  strcat(result_str, ffield);
  strcat(result_str, " ");
  }
 }
```

Step 4.: The computer will check for the existence of **fields** in the **result_str** array by calculating the size of it. If the **result_str** has any **fields** inside, its size will be bigger than one. The computer will then add a closed parenthesis (C_PARENTHES character) before the end of **result_str** array. Otherwise, the computer will write an ENDARRAY character to the beginning of **result_str** array.

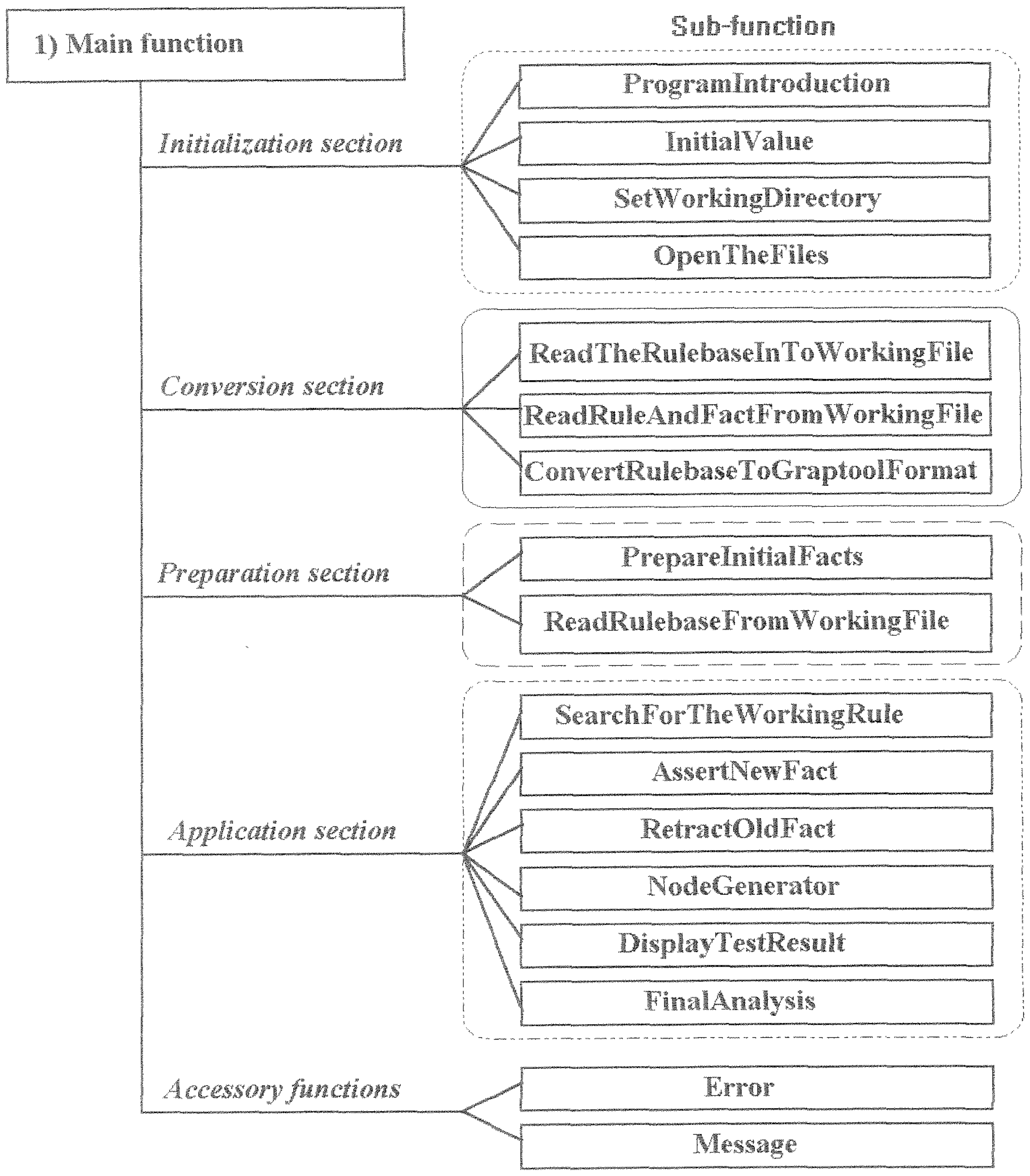
```
/* CALCULATE THE SIZE OF A result_str ARRAY */
length  = strlen(result_str);
/* CHECK THE EXISTENCE OF ANY FIELDS IN result_str ARRAY */
if(length > 1)
 result_str[length-1] = (char)C_PARENTHES;
else
 result_str[0] = ENDARRAY;
```

Step 5.: The **ReplaceVariableWithValue** will return to the calling function.

```
}
```

11.2. Major functions

There are twenty-four major functions. Sixteen of them are used in the Main function. The rest of them will be called by any of the sixteen functions.



A **Main** function is the heart of the Graptool software. The sequential work of the **Main** can be divided into four sections:

```
void main(void)
{
```

Initialization section:

Purpose: The initialization section will prepare the end user and Graptool software itself for program processing.

Pseudo-code:

Step 1.: The computer will call the **ProgramIntroduction** function to display a brief description of the Graptool software. After that, the computer will call the **InitialValue** function to initialize all the important variables.

```
/**** INITIALIZATION SECTION ****/
ProgramIntroduction();
InitialValue();
```

Step 2.: The computer will call the **SetWorkingDirectory** function to set up the working directory. After the working directory has been set up, the computer will call the **OpenTheFiles** function to open the CLIPS rule-based file, the error file (*Graptool.err)*, the working files (*Graptool.fac*, *Graptool.rul*, and *Graptool.wrk*), and the information file. The **OpenTheFiles** function also allows the end user to select the information file which will store the result of rule-based testing.

```
SetWorkingDirectory(path);
OpenTheFiles(path);
```

Conversion section:

Purpose: The conversion section is used to convert the CLIPS rule bases and initial facts into Graptool format.

Pseudo-code:

Step 3.: The computer will call the **ReadTheRulebaseInToWorkingFile** function to check for critical errors and make adjustments of the CLIPS rule-base. The result will be stored in the Graptool working file that temporarily uses the information file.

```
/**** CONVERSION SECTION ****/
ReadTheRulebaseInToWorkingFile();
```

Step 4.: The computer will first display a process message and then the computer will do the *while loop*. While it is not the end of information file, the computer will call the **ReadRuleAndFactFromWorkingFile** function to read a CLIPS rule-base from the information file (used temporarily as working file), and check for all possible critical errors. The result of this checking will be written into the *Graptool.wrk* file and the ***rule_base*** array. The computer then calls the **ConvertRulebaseToGraptoolFormat** function to convert a rule base or initial fact in the ***rule_base*** array into Graptool format. The result of this conversion will be saved in the *Graptool.rul* file or the *Graptool.fac* file. Before the computer goes back to the beginning of the *while loop*, the **delay** function is called to slow down loop processing.

```
/* DISPLAY A PROCESS MESSAGE */
Message("\n\n** Converting rule-based into Graptool format.", NONE, ENDARRAY);
/* DO GRAPTOOL FORMAT CONVERTING PROCESS */
while(!feof(iptr))
 {
  done = ReadRuleAndFactFromWorkingFile(DEFRULE);
  ConvertRulebaseToGraptoolFormat(done);
  delay(DELAY_LOOP/2);
 }
```

Step 5.: If no rule-base exists in the CLIPS file (*rule_flag* is FALSE), the computer will call the **Error** function to display an error message and stop execution. Otherwise, the computer will display the ending process message.

```
/* CHECK FOR THE EXISTENCE OF RULE BASE */
if(rule_flag == FALSE)
  Error("CANNOT FIND ANY RULES IN THIS RULE-BASE.", NONE, "\n");
Message("\n\n** Converting process is finished. \n", NONE, ENDARRAY);
```

Preparation section:

Purpose: The preparation section will prepare the initial facts and rule bases for rule-based structural testing.

Pseudo-code:

Step 6.: The computer will call the **PrepareInitialFacts** function to read the initial facts from the *Graptool.fac* file into the ***fact_base*** array. If initial facts do not exist in *Graptool.fac*, the **PrepareInitialFacts** will allow a user to enter initial facts into the ***fact_base*** array. If there are no initial facts in the ***fact_base*** (***fact_flag*** is FALSE), the computer will display a warning message. Otherwise, the computer will display a "go ahead" message. The computer will then ask the end user for confirmation of starting the Graptool rule-based testing process by entering Y or y.

```
/**** PREPARATION SECTION ****/
PrepareInitialFacts();
/* CHECK FOR THE EXISTENCE OF INITIAL FACT */
if(fact_flag == FALSE)
  Message("\n!!*****!! THE INITIAL FACTS DO NOT EXIST. !!*****!!", -1, "\n");
else
  Message("\n!!***!! EVERYTHING IS READY FOR TESTING. !!***!!",-1,"\n");
/* DECIDE WHETHER OR NOT TO START THE TESTING PROCESS */
printf("\n** Enter Y if you wish to start the rule-based testing process..> ");
gets(ans);
```

Step 7.: If the user decides to start the rule-based testing process (entering Y or y), the computer will do the procedure in step eight. Otherwise, the computer will skip to step thirteen.

```
/* CHECK IF AN USER WANTS TO CONTINUE THE PROCESSES */
if((strcmp(ans, "y") == SUCCESS) || (strcmp(ans, "Y") == SUCCESS))
 {
```

Step 8.: The computer will first clear the computer screen and set the ***assert_first*** variable to TRUE. Since the order of assert and retract is important to rule-based structural testing, the computer will ask the end user to select the applying order of assertion and retraction. If the end user wants the computer to apply the retraction first (entering N or n), the ***assert_first*** will be set to FALSE.

```
  clrscr();
  assert_first = TRUE;
  /* GET THE APPLYING ORDER OF ASSERTION AND RETRACTION */
  printf("Enter N if you want (FACT BASE - RETRACT U ASSERT),");
  printf("\nor anything else (FACT BASE U ASSERT - RETRACT)...> ");
  gets(ans);
  /* CHECK IF THE USER WANTS TO APPLY THE RETRACTION FIRST */
  if((strcmp(ans, "n") == SUCCESS) || (strcmp(ans, "N") == SUCCESS))
    assert_first = FALSE;
```

Step 9.: The computer will call the **ReadRulebaseFromWorkingFile** function to read the Graptool format rule bases from the *Graptool.rul* into a ***rule_base*** array. The computer also writes the selected assert and retract order into the information file.

```
Message("\n*** Starting rule base test ***\n", NONE, ENDARRAY);
ReadRulebaseFromWorkingFile();
if(assert_first)
 fprintf(iptr, "\n** FACT BASE SET UNION ASSERT MINUS RETRACT **\n");
else
 fprintf(iptr, "\n** FACT BASE SET MINUS RETRACT UNION ASSERT **\n");
```

Application section:

Purpose: The application section will apply the logical path algorithm to test the rule-based structure.

Pseudo-code:

Step 10.: The computer will first initialize all the variables which will be used in the testing process, such as ***notdone*** to TRUE. The computer will call the **NodeGenerator** function to generate the initial node and its condition set. The **NodeGenerator** also sets the initial node as the first working node and its condition set will be used as the facts in the fact base. The initial node and its condition set will be displayed on the screen and saved in the information file by the **DisplayTestResult** function.

```
 /***** APPLICATION SECTION *****/
/* INITIALIZE THE RULE-BASED TESTING VARIABLES */
notdone = TRUE;
display = nowhere;
ptrwork = ptrfirst;
rule_ptr = rule_base;
connect = rule_counter = fact_flag = clp_flag = SUCCESS;
NodeGenerator();
DisplayTestResult();
```

Step 11.: The computer will first check the value of ***notdone*** variable. If ***notdone*** is TRUE, the computer continues to test the rule-based structure in step 11.1 to 11.4. Otherwise, the computer will go to step 12.

```
/* DO THE TESTING RULE-BASED STRUCTURE PROCESS */
while(notdone)
```

Step 11.1. The computer copies the condition set of the working node over into the ***fact_base*** array. The **SearchForTheWorkingRule** function will be called to search the ***rule_base*** array for a rule whose conditions have been satisfied by the fact base. This rule is called a **working rule**.

```
 {
 /* INITIALIZE THE FACTS OF FACT BASE */
 strcpy(fact_base, ptrwork->condition_set);
 right_rule = SearchForTheWorkingRule(&found);
```

Step 11.2.: If the **working rule** is found (the *found* value is TRUE), rule assertions and retractions will be performed based on the selected order. After that, the **NodeGenerator** function will be called to generate the new node and its condition set and then the compute will go to step 11.4.

```
/* CHECK IF RULE LHS IS SATISFIED */
if(found)
 {
 connect = clp_flag = TRUE;
 /* CHECK THE ORDER OF ASSERTION AND RETRACTION */
 if(assert_first)
   {
    AssertNewFact(right_rule);
    RetractOldFact(right_rule);
   }
 else
   {
    RetractOldFact(right_rule);
    AssertNewFact(right_rule);
   }
 NodeGenerator();
 }
```

Step 11.3.: If the **working rule** is found (the value of *found* is FALSE), the computer will first check for the existence of a node connected to the working node. If a working node is not connected to any node (*connect* is FALSE), a warning message will be displayed. The computer then searches for a new working node that has not been used before. If a new working node is found (*ptrwork* does not equal the *nowhere* value), the variables of the testing process will be re-initialized. However, if a new working node is not found, *notdone* will be set to FALSE.

```
else
 {
  /*CHECK IF WORKING NODE CONNECTS TO ANY NODE */
  if(!connect)
    {
     printf("\n No CONNECTING NODE \n");
     fprintf(fptr, "\n No CONNECTING NODE \n");
    }
  ptrwork = ptrwork->ptrnext;
  /* SEARCH FOR THE NEXT WORKING NODE */
  while((ptrwork->duplicate != nowhere) && (ptrwork != nowhere))
        ptrwork = ptrwork->ptrnext;
   if(ptrwork != nowhere)
    {
     display = nowhere;
     rule_ptr = rule_base;
     connect = rule_counter = fact_flag = SUCCESS;
    }
   else
     notdone = FALSE;
 }
```

Step 11.4.: The computer will call the **DisplayTestResult** function to display the new node and its condition set on the screen. It will also store the result of rule-based testing to the selected information file. The computer will then go back to process step eleven.

```
    DisplayTestResult();
    }
```

Step 12.: After the computer has done the rule-based structural testing, the computer will display an ending process message and call the **FinalAnalysis** function to do simple rule-based analysis. This analysis will be shown on the screen and will be added into the information file. After that, the computer will go to step fourteen.

```
  /* DISPLAY AN ENDING PROCESS MESSAGE */
   Message("\n\n*** Rule base test is finished ***",NONE, ENDARRAY);
  FinalAnalysis();
  }
```

Step 13.: If the end user decides not to continue the rule-based testing process (user does not enter Y or y at the step seven.), a termination message will be displayed.

```
else
  Message("\n Computer processing has been terminated. \n", -1, ENDARRAY);
```

Step 14.: The computer will close every opened file, stop Graptool software execution, and return control back to the DOS operating system.

```
fcloseall();
}
```

2) ProgramIntroduction function

Purpose: A **ProgramIntroduction** function will give a general description of the Graptool software to the end user.

Pseudo-code:

The computer will continues to display the Graptool general information on the screen until a key is pressed. The **ProgramIntroduction** will then return to the **Main** function.

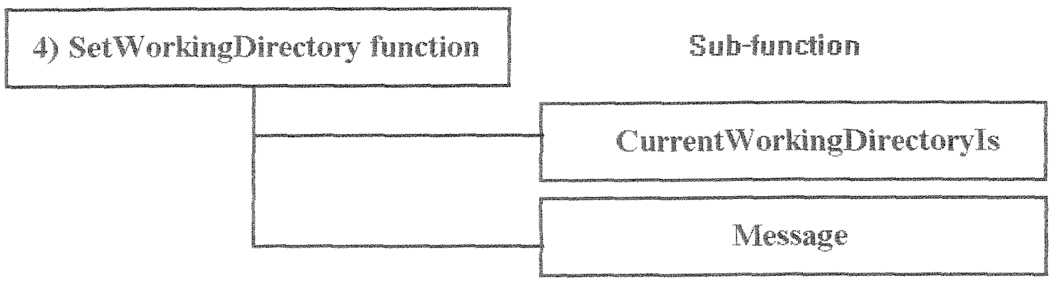
```
void ProgramIntroduction(void)
{
 int i;
 clrscr();
 for(i=0; i<79; i++) putt(205);
 printf("\n                              ******* GRAPTOOL.C *******\n");
 printf(" PROGRAM OBJECTIVE.: Graptool is a software tool based on the ");
 printf("logical path \n algorithm.  This software will read the CLIPS ");
 printf("rule base and convert it into \n Graptool format. After the ");
 printf("conversion is finished, the program will apply \n the logical path ");
 printf("algorithm to the Graptool format for testing of the rule base \n ");
 printf("structure.  The results of this tool are the following:");
 printf("\n 1. GRAPTOOL.WRK contains the CLIPS rule base after Graptool has ");
 printf("eliminated the \n    unnecessary functions or commands from the ");
 printf("original rule base.");
 printf("\n 2. GRAPTOOL.RUL contains rules from the rule base in Graptool ");
 printf("format.");
 printf("\n 3. GRAPTOOL.FAC contains initial facts from the rule base in ");
 printf("Graptool format.");
 printf("\n 4. GRAPTOOL.ERR(option) contains process and error messages ");
 printf("during software \n    execution.");
 printf("\n 5. The information file contains the results of rule base ");
 printf("testing.  The end user \n    can select any file name except the ");
 printf("CLIPS rule base file name, and the file \n    extension cannot be ");
 printf("'.CLP'.  If the end user makes an invalid name selection,    ");
 printf("Graptool will select a unique file name which starts with 'I'. \n\n";
 printf(" FINAL NOTE: All five files will be saved in the selected working ");
 printf("directory. \n");
 printf(" WARNING..: THE COMPUTER WILL AUTOMATICALLY STOP EXECUTION IF ");
 printf("IT DETECTS \n           ANY ERROR. \n");
 for(i=0; i<79; i++) putch(205);
 printf("\n\n Press any key to continue........");
 getch();
 clrscr();
}
```

3) InitialValue function

Purpose: An **InitialValue** function initializes values of important variables such as the *nowhere* ,*fact_base_size*, *rule_size*, etc.

Pseudo-code:

```
void InitialValue(void)
{
 nowhere = (struct node *)NULL;
 fact_base_size = rule_size = SUCCESS;
 rule_counter = condition_counter = TRUE;
 error_file_open = rule_flag = fact_flag = clp_flag = nocondition_flag = FALSE;
}
```



Purpose: A <u>**SetWorkingDirectory**</u> function allows the end user to set up a working directory which will be used to store the working files.

Prototype:

```
void SetWorkingDirectory(char *path)
{
```

A path is a character pointer variable which will contain an address of the selected working directory information.

Pseudo-code:

Step 1.: The computer will call the <u>**CurrentWorkingDirectoryIs**</u> function to get and display the current directory (the computer assumes the current directory to be the working directory.). The computer also asks if the end user wants to change the working directory by entering Y or y.

```
/* DO SET WORKING DIRECTORY PROCESSES */
do
 {
  /*GET A CURRENT DIRECTORY */
  CurrentWorkingDirectoryIs(path);
  /* DECIDE WHETHER OR NOT TO CHANGE A WORKING DIRECTORY */
  printf("\n Do you want to change the working directory(N)? ");
  gets(ans);
```

Step 2.: If the end user decides to change the working directory (enter Y or y), a new directory drive and a new directory path must be entered. After that, the computer will check for the existence of the new drive and the new path. If the end user makes an invalid drive selection, the working directory will not be changed. However, if the end user makes a correct drive selection but invalid directory path, the computer will select the root directory of that selection drive to be a working directory.

```
 /* CHECK IF USER WANTS TO CHANGE THE WORKING DIRECTORY */
 if((strcmp(ans, "y") == SUCCESS) || (strcmp(ans, "Y") == SUCCESS))
  {
   printf("\n!! MAXIMUM PATH INCLUDING FILE NAME IS %d BYTES.",PATH_SIZE);
   /* GET A NEW WORKING DIRECTORY DRIVE */
   printf("\n Enter the drive of new working directory(A, B, C, etc.): ");
   gets(drive);
   /* GET A NEW WORKING DIRECTORY PATH */
   printf("Enter the path of new working directory(\\ for root): ");
   gets(path);
   dptr = strupr(drive);
   disk = (int)*dptr - 65;
   setdisk(disk);
   /* CHECK FOR THE EXISTENCE OF THE GIVEN DRIVE */
   if(disk != getdisk())
     Message("\n Warning.. GIVEN DRIVE DOES NOT EXIST. \n", -1, ENDARRAY);
   else
    {
     chdir("\\");
     /* CHECK FOR THE EXISTENCE OF THE GIVEN PATH */
     if(chdir(path) != SUCCESS)
       Message("\n Warning.. GIVEN PATH DOES NOT EXIST. \n", -1, ENDARRAY);
    }
  }
```

Step 3.: The computer will keep going back to step one until the end user stops answering with a yes (entering Y or y) to the question in step one.

```
 }
/* DO PROCESSES WHILE THE USER WANTS TO CHANGE A WORKING DIRECTORY */
while((strcmp(ans, "y") == SUCCESS) || (strcmp(ans, "Y") == SUCCESS));
```

Step 4.: The **SetWorkingDirectory** returns to the **Main** function.

```
}
```

4.1) CurrentWorkingDirectory *function*

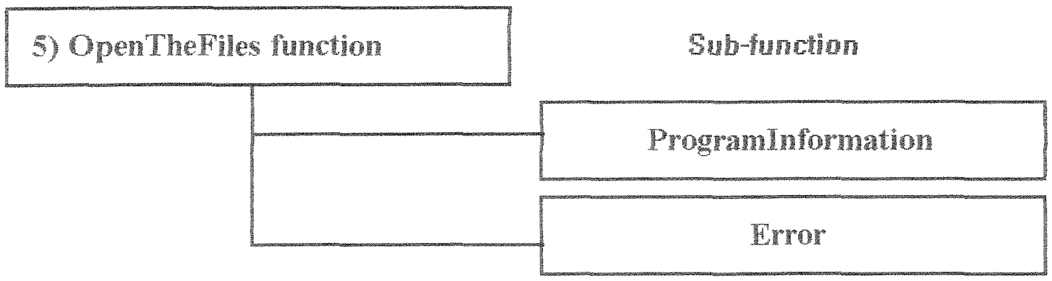
Purpose: A **CurrentWorkingDirectoryIs** function will get the current directory information.

Pseudo-code:

The computer will get the directory information from the DOS operating system and display it on the screen. The **CurrentWorkingDirectoryIs** will then return to the **SetWorkingDirectory** function.

```
void CurrentWorkingDirectoryIs(char *path)
{
 strcpy(path, "X:\\");
 path[0] = 'A' + getdisk();
 getcurdir(0, path + 3);
 printf("\n* THE CURRENT WORKING DIRECTORY -> %s \n", path);
}
```

Note: The **CurrentWorkingDirectoryIs** functions are not valid for any other operating system besides DOS. Therefore; the end user needs to modify this function when he runs the Graptool software in a different operating system.



Purpose: The <u>**OpenTheFiles**</u> function will open a CLIPS rule-based file, an information file, an error file, and working files.

Prototype:

```
void OpenTheFiles(char *path)
{
```

A path is a character pointer variable which will contain an address of the selected working directory information.

Pseudo-code:

Step 1.: The computer will get the name of the CLIPS file and the information file. The end user also needs to decide whether or not he wants to open the error file.

```
/*GET CLIPS RULE-BASE FILE */
printf("\n Enter the name of CLIPS rule base file -> ");
gets(cname);
/*GET INFORMATION FILE */
printf("Enter the name of information file(option) -> ");
gets(iname);
/* DECIDE WHETHER OR NOT TO OPEN AN ERROR FILE */
printf("Do you want to open the error file(N)? ");
gets(ans);
```

Step 2.: The computer will check the selected information file. If it is invalid, the computer will call the <u>**Error**</u> function to display an error message and stop program execution.

```
/* CONVERT FILE NAME TO CAPITAL LETTERS */
strcpy(cname, strupr(cname));
strcpy(iname, strupr(iname));
/* CHECK IF SELECTED INFORMATION FILE IS VALID*/
if((strlen(cname) != 0) && ((length = strlen(iname)) != 0))
  if((strcmp(cname, iname) == SUCCESS) ||
                                         (strcmp(&iname[length-4],".CLP") == SUCCESS))
    Error("INVALID NAME FOR THE INFORMATION FILE.", -1, "\n");
```

Step 3.: If the computer cannot open the CLIPS file, an error message will be displayed and program execution will be stopped by the <u>**Error**</u> function..

```
/* CHECK FOR THE EXISTENCE OF CLIPS RULE-BASED FILE */
if((fptr = fopen(cname, "r")) == NULL)
  Error("CANNOT OPEN THE CLIPS RULE BASE.",-1 ,"\n");
```

Step 4.: The computer opens the selected information file. If it cannot be opened, the computer will select a unique file name which starts with "'I" for the information file. Then the computer will open that file.

```
/* CHECK FOR THE EXISTENCE OF INFORMATION FILE */
if((iptr = fopen(iname, "wt+")) == NULL)
 {
 /* GET A UNIQUE INFORMATION FILE NAME */
 wname = mktemp(ifile);
  /* RE-OPEN THE INFORMATION FILE */
 iptr = fopen(wname, "wt+");
 strcpy(iname, wname);
 }
```

Step 5.: The three working files (*Graptool.rul*, *Graptool.fac*, and *Graptool.wrk)* will be opened. If the end user decides to open the error file, *Graptool.err* will be opened.

```
/* OPEN ALL THE WORKING FILES */
wptr = fopen("GRAPTOOL.WRK", "wt+");
rptr = fopen("GRAPTOOL.RUL", "wt+");
cptr = fopen("GRAPTOOL.FAC", "wt+");
/* CHECK IF USER WANTS TO OPEN AN ERROR FILE */
if((strcmp(ans, "y") == SUCCESS) || (strcmp(ans, "Y") == SUCCESS))
 {
 error_file_open = TRUE;
  /* OPEN AN ERROR FILE */
 eptr = fopen("GRAPTOOL.ERR", "wt");
 }
```

Step 6.: The computer will call the **ProgramInformation** function to display information about the initialization of Graptool software.

```
ProgramInformation(path, cname, iname);
```

Step 7.: The **OpenTheFiles** returns to the **Main** function.

```
}
```

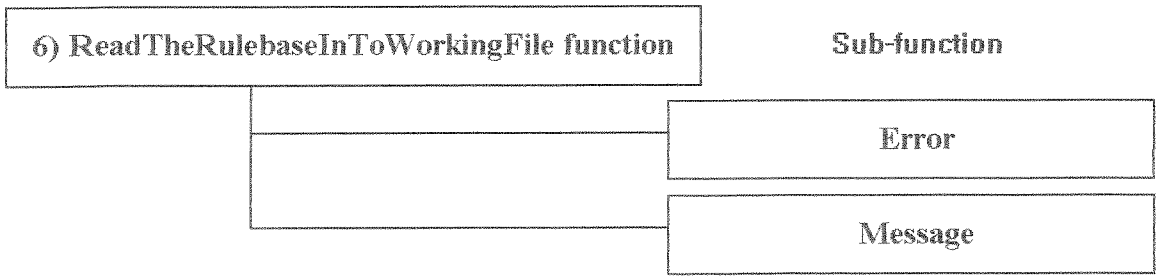
5.1) ProgramInformation function

Purpose: A **ProgramInformation** function provides information about Graptool software initialization such as the size of the fact array, the size of the field array, etc.

Pseudo-code:

The computer will displayed the information of Graptool software initialization until a key is pressed. The **ProgramInformation** will then return to the **OpenTheFiles** function.

```
void ProgramInformation(char *path, char *cname, char *iname)
{
 clrscr();
 Message("\n\n*************** PROGRAM SETUP **************,NONE, "\n");
 Message(" The fact string size is ", FACT_SIZE, " bytes.");
 Message("\n The field string size is ", FIELD_SIZE, " bytes.");
 Message("\n The rule array size is ", RULE_SIZE," bytes.");
 Message("\n The fact_base array size is ",CON_SIZE," bytes.");
 Message("\n The variable array size is ",VARIABLE_SIZE," bytes.");
 Message("\n The selected working directory is ", NONE, strupr(path));
 Message("\n The CLIPS rule base file is ", NONE, cname);
 Message("\n The information file is ",NONE, iname);
 /* CHECK IF THE ERROR FILE IS OPENED */
 if(error_file_open)
   Message("\n The error file is GRAPTOOL.ERR.", NONE, ENDARRAY);
Message("\n*************************************************", NONE, "\n");
 printf("\n Press any key to continue..........");
 getch();
 clrscr();
}
```



Purpose: A <u>**ReadTheRulebaseInToWorkingFile**</u> function will read the CLIPS rule-based expert system program eliminating unprintable characters, extra spaces and comments. It will also adjust the rule bases, and check for critical syntax errors. The results of this function will be stored in an information file which will also be used as the Graptool working file.

Prototype:

```
void ReadTheRulebaseInToWorkingFile(void)
{
```

There is no prototype in this function.

Pseudo-code:

Step 1.: The computer will first display a process message and reset the information file pointer to the beginning. The computer will then search for the first open parenthesis (indicating the beginning of a **rule-based construct**) in the CLIPS file. The result of this searching will be stored in the *ch* variable. If *ch* is not an open parenthesis (it is not a CLIPS file), the computer will call the <u>**Error**</u> function to display an error message. Otherwise, the computer will set ***result.rem***, ***parenthes_count***, and ***quote_count*** to SUCCESS.

```
Message("\n_-_ Read CLIPS rule base into the working file _-_", -1, ENDARRAY);
/* RESET INFORMATION FILE POINTER TO THE BEGINNING OF FILE */
rewind(iptr);
/* SEARCH FOR THE FIRST OPEN PARENTHESIS OF THE CLIPS FILE */
while((!feof(fptr)) && ((ch = get(fptr)) != O_PARENTHES));
/* CHECK IF IT IS AN OPEN PARENTHESIS */
if(ch != O_PARENTHES)
 Error("CANNOT FIND THE BEGINNING OF THE CLIPS RULE BASE.", NONE, "\n");
/* INITIALIZE VARIABLES VALUE */
result.rem = parenthes_count = quote_count = SUCCESS
```

Step 2.: The computer will check the *while loop* condition which is the returned value of <u>**feof(fptr)**</u> function. If <u>**feof(fptr)**</u> returns FALSE (the computer has not reach the end of CLIPS file), the computer will set acceptable_letter value to TRUE and do sub-step two. Otherwise, the computer will go to step three.

```
/* DO THE ADJUSTMENT AND CHECK SYNTAX ERROR PROCESS */
while(!feof(fptr))
 {
  /* INITIALIZE acceptable_letter  TO TRUE */
  acceptable_letter = TRUE;
```

Step 2.1.: If *ch* is not a printable character., the computer will set *acceptable_letter* to FALSE and go to step 2.10.

```
/* ELIMINATE AN UNPRINTABLE CHARACTER */
if(!isprint(ch))
 acceptable_letter = FALSE;
```

Step 2.2.: The computer will set *acceptable_letter* to FALSE and then go to step 2.10 if the *ch* value is a space, and the *previous_ch* value satisfies one of the following conditions: i) *previous_ch* is a space; ii) *previous_ch* is a first quote (*previous_ch* is a quote and *result.rem* is equal to TRUE, one,); iii) *previous_ch* is not between quotes (*result.rem* equals to SUCCESS, zero,) and its value lies in one of the logical operators (&, |, ~) or an open parenthesis; iv) *previous_ch* is not between quotes or between **block**, and is either a S_WILDCARD (?), ENDLHS (=), or RE_POINT1 (<) characters.

```
else
/* CHECK IF IT IS A SPACE */
if(ch == SPACE)
 {
  /* ELIMINATE A SPACE AFTER ANY SPACE */
  if(previous_ch == SPACE)
   acceptable_letter = FALSE;
  else
  /* ELIMINATE A SPACE AFTER THE FIRST QUOTE */
  if((previous_ch == QUOTE) && (result.rem == TRUE))
    acceptable_letter = FALSE;
  else
  /* CHECK IF IT IS OUTSIDE THE QUOTE */
  if(result.rem == SUCCESS)
   {
    /* ELIMINATE A SPACE AFTER ALL THESE CHARACTERS */
    if((previous_ch == LOGI_OR) || (previous_ch == LOGI_AND) ||
       (previous_ch == LOGI_NOT) || (previous_ch == O_PARENTHES))
     acceptable_letter = FALSE;
    else
    if(((previous_ch == S_WILDCARD) || (previous_ch == RE_POINT1) ||
        (previous_ch == ENDLHS)) && (parenthes_count == TRUE))
     acceptable_letter = FALSE;
   }
 }
```

Step 2.3.: If *ch* is a COMMENT character (indicating the beginning of the CLIPS comment), the computer will skip the entire comment (*ch* is set to NEXTLINE character), set *acceptable_letter* to FALSE, and go to step 2.10.

```
else
/* CHECK IF IT IS THE BEGINNING OF THE COMMENT */
if((ch == COMMENT) && (result.rem == SUCCESS))
{
 /* SEARCH FOR THE END OF A COMMENT */
 while((!feof(fptr)) && ((ch = getc(fptr)) != NEXTLINE));
 acceptable_letter = FALSE;
}
```

Step 2.4.: If *ch* is a quote and it is outside any **block** (*parenthes_count* is TRUE), the computer will treat it as the beginning of a comment which occurs after the **rule-based construct** name. The computer will then skip over the whole comment, set *acceptable_letter* to FALSE, and go to step 2.10.

```
else
/* CHECK IF IT IS A COMMENT AFTER THE RULE-BASED CONSTRUCT NAME */
if((ch == QUOTE) && (parenthes_count == TRUE))
 {
 /* SEARCH FOR THE END OF COMMENT */
 while((!feof(fptr)) && ((ch = getc(fptr)) != QUOTE));
 acceptable_letter = FALSE;
 }
```

Step 2.5.: If *ch* is a quote inside a **block** (*parenthes_count* is more than one), the computer will re-calculate the *ch* position, and add a space to the information file if *ch* is the first quote (*result.rem* equal to TRUE) and *previous_ch* is neither an open parenthesis nor a space. Then it will go to step 2.10.

```
else
/* CHECK IF IT IS A QUOTE INSIDE A BLOCK */
if((ch == QUOTE) && (parenthes_count > 1))
 {
 result = div((++quote_count), 2);
 /* ADD A SPACE TO WORKING FILE  IF IT IS A FIRST QUOTE AND */
 /* AFTER A CHARACTER BESIDE AN OPEN PARENTHESIS AND A SPACE */
 if((result.rem == TRUE) &&
     (previous_ch != O_PARENTHES) && (previous_ch != SPACE))
   putc(SPACE, iptr);
 }
```

Step 2.6.: If *ch* is either S_WILDCARD (?), ENDLHS (=), RE_POINT1 (<) character, or is outside a **block** (*parenthes_count* equal to TRUE), and *previous_ch* is not a space, the computer will add a space to the information file and then go to step 2.10.

```
else
/* ADD A SPACE IN FRONT OF THESE CHARACTERS  IF IT IS OUTSIDE FACT BLOCK*/
if(((ch == S_WILDCARD) || (ch == ENDLHS) || (ch == RE_POINT1)) &&
                          (previous_ch != SPACE) && (parenthes_count == TRUE))
 putc(SPACE, iptr);
```

Step 2.7.: If *ch* is an open parenthesis and outside the quotes (*result.rem* equal to SUCCESS), the computer will add 1 to the *parenthes_count* value, and a space to the information file if *previous_ch* variable is neither a space nor an open parenthesis. Then it will go to step 2.10.

```
else
/* CHECK IF IT IS AN OPEN PARENTHESIS OUTSIDE THE QUOTES */
if((ch == O_PARENTHES) && (result.rem == SUCCESS))
 {
 parenthes_count++;
 /* ADD A SPACE IN FRONT OF AN OPEN PARENTHESIS */
 if((previous_ch != SPACE) && (previous_ch != O_PARENTHES))
  putc(SPACE, iptr);
 }
```

Step 2.8.: If *ch* is a closed parenthesis outside the quotes (*result.rem* equal to SUCCESS), the computer will subtract 1 from the *parenthes_count* value and go to step 2.10.

```
else
/* SUBTRACT ONE FROM parenthes_count FOR CLOSE PARENTHESIS */
if((ch == C_PARENTHES) && (result.rem == SUCCESS))
 parenthes_count--;
```

Step 2.9.: If the *parenthes_count* value is equal to SUCCESS (the *ch* is outside a **rule-based construct**) and *ch* is neither an open parenthesis nor a space, the computer will call the **Error** function to display an error message.

```
else
/* CHECK IF IT IS AN UNIDENTIFIED CHARACTER */
if((parenthes_count == SUCCESS) && (ch != O_PARENTHES) && (ch != SPACE))
 Error("FIND UNIDENTIFIED CHARACTERS IN THE CLIPS RULE BASE.", NONE, "\n");
```

Step 2.10.: If *acceptable_letter* is TRUE (a character inside the *ch* variable is acceptable), the computer will write the *ch* value into the information file and the *previous_ch* variable. After that, the computer will get the new character from the CLIPS file and store it in the *ch* variable and return to step two.

```
/* WRITE AN ACCEPTABLE CHARACTER INTO THE INFORMATION FILE */
if(acceptable_letter)
 {
 putc(ch, iptr);
 previous_ch = ch;
 }
/* GET A NEW CHARACTER FROM CLIPS FILE */
ch = getc(fptr);
}
```

Step 3.: The computer will check for missing parentheses. If a parentheses is missing (*parenthes_count* is not equal to SUCCESS), the computer will send an error message to the **Error** function.

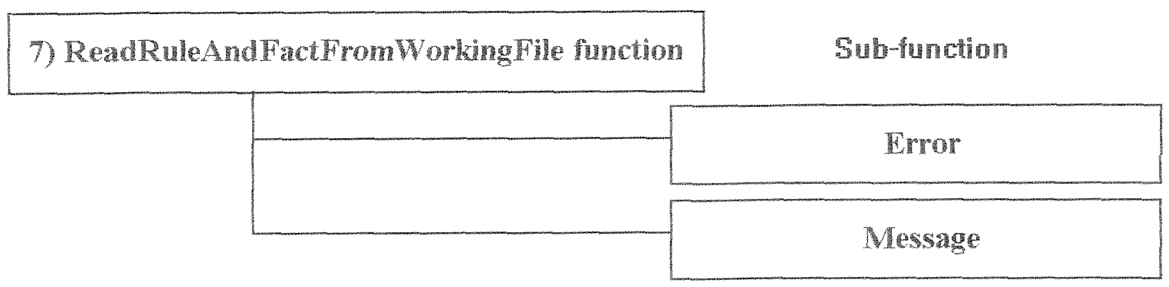
```
/* CHECK FOR MISSING PARENTHESIS */
if(parenthes_count != SUCCESS)
 Error("MISSING THE PARENTHESIS IN THE CLIPS RULE BASE.", NONE, "\n");
```

Step 4.: The computer will close the CLIPS file, reset the information file pointer to the beginning, and display the ending message process.

```
/* CLOSING A CLIPS FILE */
fclose(fptr);
/* SET THE POINTER OF INFORMATION FILE TO THE BEGINNING */
rewind(iptr);
Message("\n\n _-_-_- Complete the reading _-_-_-_", NONE, "\n");
```

Step 5.: The **ReadTheRulebaseInToWorkingFile** will return to the **Main** function.

```
}
```



Purpose: A <u>**ReadRuleAndFactFromWorkingFile**</u> function will read one **rule-based construct** at a time from the information file into a ***rule_base*** array. The computer will make adjustments and check for critical errors which it did not do in the <u>**ReadTheRulebaseInToWorkingFile**</u> function. The computer also compares Graptool's software configuration with its needs. If the configuration does not match its need or any errors have been detected, an appropriate error message will be displayed on the screen. The result of this function will be stored in the *Graptool.wrk* file and ***rule_base*** array.

Prototype:

```
int ReadRuleAndFactFromWorkingFile(int type)
{
```

A type is an integer variable which will be a DEFFACTS or DEFRULE character. The DEFFACTS character will indicate to the computer that it only needs to read a **deffacts construct**. However, the DEFRULE character will indicate that only the **rule-based construct** (defrule and deffacts construct) needs to be read..

Pseudo-code:

Step 1.: The computer will first assign rule_base value to *currptr*, set *limit* to RULE_SIZE minus two, and set *rule_ptr*, *parenthes_count*, *quote_count*, *result.rem* and *done* to SUCCESS (zero). The computer will then search for an open parenthesis in the information file and save that result in the *ch* variable. If *ch* is an open parenthesis (the beginning of **rule-based construct** is found), the computer will do step two. Otherwise, the computer will go to step six.

```
/* INITIALIZE VARIABLES VALUE */
currptr = rule_base;
limit = RULE_SIZE - 2;
rule_ptr = parenthes_count = quote_count = result.rem = done = SUCCESS;
/* SEARCH FOR THE BEGINNING OF RULE-BASED CONSTRUCT */
while((!feof(iptr)) && ((ch = getc(iptr)) != O_PARENTHES));
/* CHECK IF IT IS AN OPEN PARENTHESIS (BEGINNING OF CONSTRUCT) */
if(ch == O_PARENTHES)
```

Step 2.: The computer will start reading the **rule-based construct** into the ***rule_base*** array process by checking if the character being read (*ch* variable) is outside the quote. If the *result.rem* is equal to SUCCESS (*ch* is outside the quote), the computer will do step 2.1. Otherwise, the computer will go to step 2.2.

```
{
/* DO READING OF A RULE-BASED CONSTRUCT PROCESS */
do
 {
 /* CHECK IF ch ( BEING READ CHARACTER) IS OUTSIDE QUOTES */
 if(result.rem == SUCCESS)
```

Step 2.1.: The computer will do the following:

The computer will update the *parenthes_count* value. If *ch* is an open parenthesis, the computer will add 1 to the *parenthes_count*. However, if *ch* is a closed parenthesis, the computer will subtract 1 from the *parenthes_count*.

```
 {
 /* ADD ONE TO parenthes_count FOR AN OPEN PARENTHESIS */
 if(ch == O_PARENTHES)
  parenthes_count++;
 else
 /* SUBTRACT ONE FROM parenthes_count FOR A CLOSED PARENTHESIS */
 if(ch == C_PARENTHES)
  parenthes_count--;
```

The computer will add a space behind the last quote (*ch* will be added behind) in the ***rule_base*** array if *ch* is not a space or a closed parenthesis.

```
 /* ADD A SPACE BEHIND  LAST QUOTE */
 if((ch != SPACE) &&
        (ch != C_PARENTHES) && (rule_base[rule_ptr-1] == QUOTE))
  {
   rule_base[rule_ptr] = SPACE;
   rule_ptr++;
  }
```

The computer will add a space in front of "<-" if *ch* is a RE_POINT2 character(-) that follows RE_POINT1 character (<), and the character in front of a RE_POINT1 is not a space.

```
 /* ADD A SPACE IN FRONT OF "<-" */
 if((rule_base[rule_ptr-1] == RE_POINT1) &&
          (ch == RE_POINT2) && (rule_base[rule_ptr-2] != SPACE))
  {
  rule_base[rule_ptr-1] = SPACE;
  rule_base[rule_ptr] = RE_POINT1;
  rule_ptr++;
  }
```

The computer will eliminate a space in front of *ch* if it is a closed parenthesis, a logical **and**, or a logical **or**.

```
 /* DELETE SPACE BEFORE A CLOSE PARENTHESIS AND LOGICAL OPERATORS */
 if(((ch == C_PARENTHES) || (ch == LOGI_AND) || (ch == LOGI_OR)) &&
                                        (rule_base[rule_ptr-1] == SPACE))
  rule_ptr--;
```

The computer will replace the **arrow**, =>, with a LHSRHS character and then go to step 2.3.

```
/* REPLACE A CLIPS ARROW WITH A LHSRHS CHARACTER */
if((parenthes_count == TRUE) && (ch == STARTRHS) &&
                                        (rule_base[rule_ptr-1] == ENDLHS))
  {
   rule_ptr--;
   ch = LHSRHS;
  }
}
```

Step 2.2.: The computer will check if *ch* is between quotes. If *ch* is inside quotes (result.rem is TRUE), its value will be replaced with Q_SPACE, O_PAREN, or C_PAREN depending on whether *ch* is a space, an open parenthesis, or a closed parenthesis, respectively.

```
else
/* CHECK IF ch IS BETWEEN QUOTES */
if(result.rem == TRUE)
 {
 /* REPLACE SPACE WITH Q_SPACE CHARACTER */
 if(ch == SPACE)
   ch = Q_SPACE;
 else
 /* REPLACE OPEN PARENTHESIS WITH O_PAREN CHARACTER*/
 if(ch == O_PARENTHES)
   ch = O_PAREN;
 else
 /* REPLACE CLOSED PARENTHESIS WITH C_PAREN CHARACTER */
 if(ch == C_PARENTHES)
   ch = C_PAREN;
 }
```

Step 2.3.: The computer will check if *ch* is a quote. If it is, the computer will update the *quote_count* value and re-calculate the *result.rem* value. If *ch* is the last quote (*result.rem* is equal to SUCCESS) following a space, the computer will eliminate that space.

```
/* CHECK IF ch IS A QUOTE */
if(ch == QUOTE)
 {
 /* RE-CALCULATE result.rem VALUE */
 result = div((++quote_count), 2);
 /* ELIMINATE A SPACE BEFORE THE LAST QUOTE */
 if((result.rem == SUCCESS) && (rule_base[rule_ptr-1] == Q_SPACE))
   rule_ptr--;
 }
```

Step 2.4.: The computer will add the *ch* value into a ***rule_base*** array, get the next character (store in *ch* variable) from the information file, and increment *rule_ptr* value by one. It will then go back to perform step two if the end of the information file has not been reached (**feof(iptr)** is FALSE), the *parenthes_count* value does not equal SUCCESS (it is not the end of **rule-based construct**), and there is not an excess in the size of the ***rule_base*** array.

```
 /* WRITE ch VALUE TO rule_base ARRAY */
 rule_base[rule_ptr] = ch;
 /* GET A NEXT CHARACTER */
 ch = getc(iptr);
/* INCREMENT rule_ptr VALUE BY ONE */
 rule_ptr++;
 }
while((!feof(iptr)) && (parenthes_count != SUCCESS) && (rule_ptr < limit));
rule_base[rule_ptr] = ENDARRAY;
```

Step 3.: Before the computer can determine what type of **rule-based construct** (defrule or deffacts construct) is in the ***rule_base*** array, it must check the *parenthes_count* value. If the *parenthes_count* value does not equal SUCCESS (the reading process is stopped before the computer reaches the end of the **rule-based construct** being read), the computer will call an **Error** function to display an error message (assuming that the ***rule_base*** array is too small for a complete **rule-based construct**). Otherwise, the computer will search for the end of a **first field** (a space) in the ***rule_base*** and its address will be stored in the *currptr* variable. If a space isn't found (**currptr* value is an ENDARRAY character), the computer will call **Error** function to display an error message (assuming that the **rule-based construct** has nothing). Otherwise, the **currptr* value (a space) will be replaced by an ENDARRAY character.

```
/* CHECK IF THE END OF RULE-BASED CONSTRUCT HAS NOT BEEN READ */
if(parenthes_count != SUCCESS)
  Error("THE RULE ARRAY IS TOO SMALL. ", NONE, "\n");
/* SEARCH FOR THE END OF THE FIRST FIELD */
currptr = strchr(rule_base, SPACE);
/* CHECK FOR THE EXISTENCE OF THE FIRST FIELD */
if(*currptr == ENDARRAY)
  Error("CANNOT FIND ANYTHING IN THE CLIPS RULE BASE. ", NONE, "\n");
*currptr = ENDARRAY;
```

Step 4.: The computer will know what type of **rule-based construct** is in the ***rule-base*** array by determining the first field. If the **first field** is a "defrule", *fact* variable will be set to FALSE (**rule-based construct is a defrule construct**). Otherwise, if **first field** is a "deffacts", *fact* variable will be set to TRUE (**rule-based construct** is a **deffacts construct**). However, if neither "defrule" nor "deffacts" are the **first field**, the computer will display an error message.

```
/* CHECK IF IT IS A DEFRULE CONSTRUCT */
if(strcmp(&rule_base[1], "defrule") == SUCCESS)
 fact = FALSE;
else
/* CHECK IF IT IS A DEFFACTS CONSTRUCT */
if(strcmp(&rule_base[1], "deffacts") == SUCCESS)
 fact = TRUE;
else
 Error("CANNOT FIND DEFRULE OR DEFFACTS IN THE RULE BASE.", NONE, "\n");
```

Step 5.: Once the computer knows the rule-based construct type, it will check the value of the **type** variable. If **type** is DEFFACTS and ***rule_base*** is defrule construct (*fact* is FALSE), the computer will set the *done* to NONE (the computer did not read the right construct) and go to step six. Otherwise, the computer will replace a **currptr* value (an ENDARRAY character) with its original value (a space). It will then assign the *currptr* value, incremented by one (beginning address of rule-based construct name), to the *temptr*. The computer will then do sub-step five.

```
/* CHECK IF COMPUTER READS THE RIGHT CONSTRUCT */
if((type == DEFFACTS) && (!fact))
 done = NONE;
else
  {
  *currptr = SPACE;
   temptr = ++currptr;
```

Step 5.1.: The computer will search for the rule-based construct name starting from *currptr* value. The end of a rule-based construct name can be identified as follows: i) If ***rule_base*** is a **defrule construct** and it does not have any LHS pattern, a LHSRHS character will be found after a name: ii). If a name is followed by a **fact block**, an open parenthesis will indicate the end of that name; iii). A space can be found after a name because each **field** is normally separated by a space; iv). If a ***rule_base*** is a **defrule construct** and a name is followed by an **index retract**, a question mark will indicate the end of that name; v). It is possible that this ***rule_base*** does not have anything after **the first field**. An ENDARRAY character (\0) will be used as a terminating character, meaning that the computer will stop a searching loop whenever it reads an ENDARRAY.

If the computer cannot find a name (*currptr* still equal to *temptr* or **currptr* is an ENDARRAY character), an error message will be displayed. Otherwise, the computer will save the **currptr* value in the *ch*, replace **currptr* value with an ENDARRAY character, and display the rule-based construct name with its type on the screen. After that, the computer will replace **currptr* value with the previous *ch* value.

```
/* SEARCH FOR THE ENDING OF rule_base ARRAY NAME */
while((*currptr != LHSRHS) && (*currptr != O_PARENTHES) &&
 (*currptr != SPACE) && (*currptr != S_WILDCARD) && (*currptr != '\0'))
  currptr++;
/* CHECK FOR THE EXISTENCE OF rule_base ARRAY NAME */
if((currptr == temptr) || (*currptr == ENDARRAY))
 Error("CANNOT FIND DEFRULE NAME OR DEFFACTS NAME. ", NONE, "\n");
ch = *currptr;
*currptr = ENDARRAY;
/* DISPLAY A rule_base ARRAY NAME WITH ITS TYPE */
Message("\n\n FINISH READING.....",NONE, &rule_base[1]);
*currptr = ch;
```

Step 5.2.: The computer will check if ***rule_base*** is a **defrule construct**. If it is a **defrule construct** (*fact* is FALSE), the computer will update a rule_size value. The computer will then initialize all the variables before it checks the configuration and syntax error procedures in step 5.3. This procedure includes writing a **rule-based construct** from the ***rule_base*** array into a *Graptool.wrk* file.

```
/* CHECK IF IT IS A DEFRULE CONSTRUCT */
if(!fact)
  rule_size = rule_size + rule_ptr;
/* INITIALIZE VARIABLES VALUE */
quote_count = space_count = variable_count = SUCCESS;
rule_ptr = lhsrhs = result.rem = fact_size = field_size = SUCCESS;
```

Step 5.3 The computer will check the *while loop* condition, which is a rule_base[rule_ptr] value. If rule_base[rule_ptr] is not an ENDARRAY character, the computer will do sub-step 5.3. Otherwise, the computer will go to step 5.4.

```
/* DO A CHECKING CONFIGURATION AND SYNTAX ERROR PROCESS */
while(rule_base[rule_ptr] != ENDARRAY)
```

Step 5.3.1.: If a character inside rule_base[rule_ptr] is a space, the computer will do the following: i) Add a space into *Graptool.wrk* and calculate the position of that space (*result.rem* value); ii) If *result.rem* value is TRUE (it is a space in front of a **field**), the computer will assign the *rule_ptr* value to *field_start*. Otherwise, the computer will re-calculate the maximum size of the field in the ***rule_base*** array; iii) Go to step 5.3.7.

```
{
/* CHECK IF IT IS A SPACE */
if(rule_base[rule_ptr] == SPACE)
 {
  /* WRITE A SPACE TO Graptool.wrk FILE */
  putc(SPACE, wptr);
  /* CALCULATE THE SPACE POSITION */
  result = div((++space_count), 2);
  /* CHECK IF A SPACE IS IN FRONT OF A FIELD */
  if(result.rem == TRUE)
    field_start = rule_ptr;
  else
  /* CALCULATE THE MAXIMUM SIZE OF FIELD */
  if(field_size < (rule_ptr - field_start + 1))
    field_size = rule_ptr - field_start + 1;
 }
```

Step 5.3.2.: If a character inside rule_base[rule_ptr] is a quote, the computer will increment *quote_count* value by 1, add a quote into *Graptool.wrk*, and go to step 5.3.7.

```
else
/* CHECK  IF IT IS A QUOTE */
if(rule_base[rule_ptr] == QUOTE)
 {
  /* ADD ONE TO quote_count VALUE */
  quote_count++;
  /* ADD A QUOTE TO Graptool.wrk */
  putc(QUOTE, wptr);
 }
```

Step 5.3.3.: If a character inside rule_base[rule_ptr] is a S_WILDCARD character (assuming it is part of **variable field**), the computer will do the following: i) If a character inside rule_base[rule_ptr} is also part of **index retract**, the computer will add a linefeed (NEXTLINE character) and a space to *Graptool.wrk* file; ii) The computer will add one to ***variable_count*** (update number of variable file), and write a S_WILDCARD character into *Graptool.wrk*; iii) The computer will go to step 5.3.7.

```
else
/* CHECK IF IT IS A PART OF VARIABLE FIELD */
if(rule_base[rule_ptr] == S_WILDCARD)
 {
  /* CHECK IF IT IS A PART OF RETRACT VARIABLE */
  if((parenthes_count == 1) && (rule_base[rule_ptr-2] != C_PARENTHES))
   {
    /* ADD A LINEFEED AND A SPACE TO Graptool.wrk FILE */
        putc(NEXTLINE, wptr);
        putc(SPACE, wptr);
   }
  /* UPDATE A NUMBER OF VARIABLE FIELD */
  variable_count++;
  /* ADD A QUESTION MARK TO Graptool.wrk FILE */
  putc(S_WILDCARD, wptr);
 }
```

Step 5.3.4.: If a rule_base[rule_ptr] is an open parenthesis, the computer will do the following: i) Add one to ***parenthes_count*** and store ***rule_ptr*** value (beginning of a block) in the ***fact_start*** variable; ii) A linefeed and a space will be added to *Graptool.wrk* if this open parenthesis is followed by an **index retract** and/or **block**; iii) Add an open parenthesis to *Graptool.wrk*; iv) the computer will go to step 5.3.7.

```
else
/* CHECK IF IT IS AN OPEN PARENTHESIS */
if(rule_base[rule_ptr] == O_PARENTHES)
 {
  /* ADD ONE TO parenthes_count */
  parenthes_count++;
  fact_start = rule_ptr;
  /* CHECK IF IT IS THE BEGINNING OF FACT BLOCK */
  if((parenthes_count == 2) && (rule_base[rule_ptr-2] != RE_POINT2)
                    && (rule_base[rule_ptr-2] != C_PARENTHES))
   {
    /* ADD A LINEFEED AND A SPACE TO Graptool.wrk FILE */
        putc(NEXTLINE, wptr);
        putc(SPACE, wptr);
   }
  /* ADD AN OPEN PARENTHESIS TO Graptool.wrk */
  putc(O_PARENTHES, wptr);
 }
```

Step 5.3.5.: If a rule_base[rule_ptr] is a closed parenthesis, the computer will do the following: i) Subtract 1 from *parenthes_count* and add a closed parenthesis to *Graptool.wrk*; ii) Re-calculate the maximum size of a **block** in the ***rule_base*** array; iii) If the closed parenthesis is not followed by another closed parenthesis, the computer will add a linefeed to *Graptool.wrk*, iv) The computer will go to step 5.3.7.

```
else
/* CHECK IF IT IS A CLOSED PARENTHESIS */
if(rule_base[rule_ptr] == C_PARENTHES)
 {
  parenthes_count--;
  putc(C_PARENTHES, wptr);
  /* RE-CALCULATE THE MAXIMUM FACT SIZE */
  if(fact_size < (rule_ptr - fact_start + 1))
         fact_size = rule_ptr - fact_start + 1;
  /* CHECK IF IT IS FOLLOWED BY ANOTHER CLOSED PARENTHESIS */
  if(rule_base[rule_ptr + 1] != C_PARENTHES))
    /* ADD A LINEFEED TO Graptool.wrk */
         putc(NEXTLINE, wptr);
 }
```

Step 5.3.6.: The computer will do the following: i) If a rule_base[rule_ptr] is a LHSRHS character, the computer will set *lhsrhs* to TRUE and write an **arrow** (=>) into *Graptool.wrk*; ii) If a rule_base[rule_ptr] is a Q_SPACE, the computer will add a space to the *Graptool.wrk*; iii) If a character in rule_base[rule_ptr] is the O_PAREN, an open parenthesis will be added to *Graptool.wrk*, iv) If the value of rule_base[rule_ptr] is a C_PAREN, a closed parenthesis will be added to *Graptool.wrk*. Otherwise, a character in rule_base[rule_ptr] will be written into *Graptool.wrk*; v) The computer will go to step 5.3.7.

```
else
 /* REPLACE A LHSRHS CHARACTER WITH => IN Graptool.wrk*/
 if(LHSRHS = rule_base[rule_ptr])
      {
        lhsrhs = TRUE;
        putc(ENDLHS, wptr);
        putc(STARTRHS, wptr);
      }
else
/*REPLACE A Q_SPACE WITH A SPACE IN Graptool.wrk */
if(rule_base[rule_ptr] == Q_SPACE)
      putc(SPACE, wptr);
else
/*REPLACE  AN O_PAREN WITH AN OPEN PARENTHESIS IN Graptool.wrk */
if(rule_base[rule_ptr] == O_PAREN)
      putc(O_PARENTHES, wptr);
else
/*REPLACE  A C_PAREN WITH A CLOSED PARENTHESIS IN Graptool.wrk */
if(rule_base[rule_ptr] == C_PAREN)
      putc(C_PARENTHES, wptr);
else
 /* WRITE A CHARACTER BEING READ INTO Graptool.wrk */
      putc(rule_base[rule_ptr], wptr);
```

Step 5.3.7.: The computer will increment the *rule_ptr* value by 1 and then go back to step 5.3.

```
    /* INCREMENT A rule_ptr BY ONE */
    rule_ptr++;
   }
```

Step 5.4.: The computer will show an error message and stop program execution if one of the following error conditions occur: 1) An **arrow** is found in the **deffacts construct**, 2) No **arrow** is found in the **defrule construct**, 3) There is a missing quote, 4) The size of ***rule_base***, ***fact_base***, ***field***, or ***variable*** array is too small.

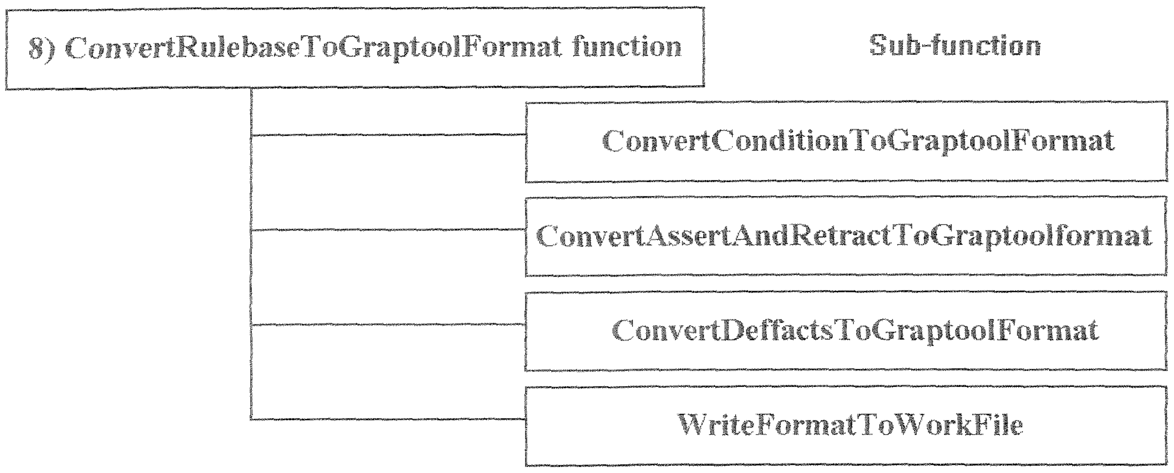
```
  /* ADD A LINEFEED TO GRAPTOOL.WRK */
  putc(NEXTLINE, wptr);
  /*CHECK IF AN ARROW IS IN DEFFACTS CONSTRUCT */
  if(fact && lhsrhs)
   Error("FIND => IN THE DEFFACTS." ,NONE, "\n");
  /* CHECK IF THERE IS NO ARROW IN DEFRULE CONSTRUCT */
  if(!fact && !lhsrhs)
   Error("CANNOT => IN THE DEFRULE.", NONE, "\n");
  result = div(quote_count, 2);
  /* CHECK FOR MISSING QUOTES */
  if(result.rem != SUCCESS)
   Error("MISSING QUOTES IN THE CLIPS RULE BASE.", NONE, "\n");
  /* CHECK IF A SIZE OF rule_base ARRAY IS TOO SMALL */
  if(rule_size >= limit)
   Error("RULE_SIZE, ",RULE_SIZE, ", BYTES IS TOO SMALL. \n");
  /* CHECK IF A SIZE OF fact_base ARRAY IS TOO SMALL */
  if(fact_size >= FACT_SIZE-2)
   Error("FACT_SIZE, ",FACT_SIZE, ", BYTES IS TOO SMALL. \n");
  /* CHECK IF A SIZE OF field ARRAY IS TOO SMALL */
  if(field_size >= FIELD_SIZE-2)
   Error("FIELD_SIZE, ",FIELD_SIZE, ", BYTES IS TOO SMALL. \n");
  /* CHECK IF A SIZE OF variable ARRAY IS TOO SMALL */
  if((variable_count*(field_size + fact_size)) >= VARIABLE_SIZE-2)
   Error("VARIABLE_SIZE, ",VARIABLE_SIZE,", BYTES IS TOO SMALL. \n");
```

Step 5.5.: The computer will display a "no error" message and initialize a *done* value to a DEFRULE. However, if the ***rule_base*** array is a **deffacts construct** (*fact* is TRUE), the computer will set *done* to DEFFACTS.

```
   Message("\n** DOES NOT DETECT ANY ERROR IN READING PROCESS **",-1,"\0");
   done = DEFRULE;
  /* CHECK IF IT IS A DEFFACTS CONSTRUCT */
   if(fact)
    done = DEFFACTS;
  }
 }
```

Step 6.: The <u>ReadRuleAndFactFromWorkingFile</u> will return to the <u>Main</u> function with a *done* value (DEFRULE, DEFFACTS, or, NONE).

```
return(done);
}
```


8) ConvertRulebaseToGraptoolFormat function
Sub-function
ConvertConditionToGraptoolFormat
ConvertAssertAndRetractToGraptoolformat
ConvertDeffactsToGraptoolFormat
WriteFormatToWorkFile

Purpose: A <u>**ConvertRulebaseToGraptoolFormat**</u> function will convert a **defrule construct** or a **deffacts construct** in the ***rule_base*** array into Graptool format. The CLIPS **defrule**, **deffacts**, **assert**, and **retract** commands will be replaced by DEFRULE, DEFFACTS, ASSERT, and RETRACT characters, respectively. Open and closed parentheses will be used as the determiner for the beginning and end of each **block**, and as a name of that **rule-based construct**. Each rule condition begins with a CONDITION character followed by &, |, and ~, to represent the logical operator of each condition. The end of each defrule and deffacts construct will be marked by an ENDRF character, for separating one construct from another.

Prototype:

```
void ConvertRulebaseToGraptoolFormat(int rulebase_type)
{
```

A rulebase_type is an integer variable which contains the **rule-based construct** in the ***rule_base*** array. The value of rulebase_type (NONE, DEFFACTS, or DEFRULE.) is the returning value of the <u>**ReadRuleAndFactFromWorkingFile**</u> function.

Pseudo-code:

Step 1.: The computer will check for the value of rulebase_type. If rulebase_type is not NONE (negative one), the computer will search for the beginning and end of the **rule-based construct** name, and add an open and closed parentheses around it. The computer will then perform either step 1.1 or step 1.2 based on the value of rulebase_type. However, if rulebase_type is NONE, the computer will go to step two.

```
/* CHECK IF IT IS NOT NONE */
if(rulebase_type != NONE)
 {
 /* SEARCH FOR THE BEGINNING OF RULE-BASED CONSTRUCT */
 /* NAME AND REPLACE IT WITH AN OPEN PARENTHESIS */
 temptr = strchr(rule_base, SPACE);
 *temptr = O_PARENTHES;
 /* SEARCH FOR THE END OF RULE-BASED CONSTRUCT */
 /* NAME AND REPLACE IT WITH A CLOSED PARENTHESIS */
 rulebase_ptr = strchr(temptr, SPACE);
 *rulebase_ptr = C_PARENTHES;
 rulebase_ptr++;
```

Step 1.1.: If the rulebase_type is a DEFRULE character, a rule_flag variable will be set to TRUE and the following functions will be called in sequential order:

The **WriteFormatToWorkingFile** function is called to write the name of **defrule construct** into the *Graptool.rul* file.

The **ConvertConditionToGraptoolFormat** function is called to convert the LHS of the rule into Graptool format.

The **ConvertAssertAndRetractToGraptoolFormat** function is called to convert the **assert block** and **retract block** of the rule into Graptool format.

The computer will go to step two.

```
/* CHECK IF IT IS A DEFRULE CONSTRUCT */
if(rulebase_type == DEFRULE)
 {
 rule_flag = TRUE;
 /* WRITE A DEFRULE CONSTRUCT NAME TO Graptool.rul */
 WriteFormatToWorkingFile(DEFRULE, temptr, rptr);
 /* CONVERT RULE LHS INTO GRAPTOOL FORMAT */
 ConvertConditionToGraptoolFormat(rulebase_ptr);
 /* CONVERT ASSERT AND RETRACT BLOCKS INTO GRAPTOOL FORMAT */
 ConvertAssertAndRetractToGraptoolFormat(rulebase_ptr);
 }
```

Step 1.2.: If the rulebase_type is a DEFFACTS character, the computer will call the following functions in sequential order:

The **WriteFormatToWorkingFile** function is called to write the name of **deffacts construct** to the *Graptool.fac* file.

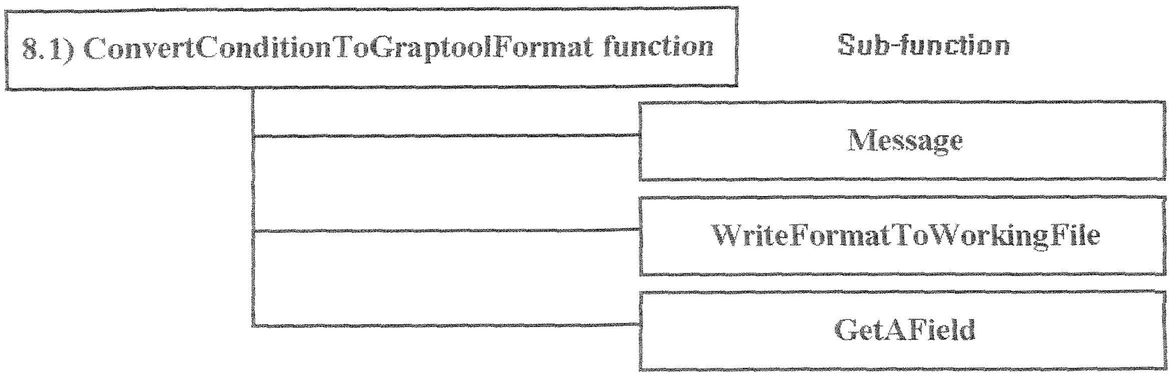
The **ConvertDeffactsToGraptoolFormat** function is called to convert the **fact blocks in deffacts construct** into Graptool format.

The computer will go to step two.

```
else
/* CHECK IF IT IS A DEFFACTS CONSTRUCT */
if(rulebase_type == DEFFACTS)
 {
 /* WRITE A DEFFACTS CONSTRUCT NAME TO Graptool.fac */
 WriteFormatToWorkingFile(DEFFACTS, temptr, cptr);
 /* CONVERT FACT BLOCKS TO GRAPTOOL FORMAT*/
 ConvertDeffactsToGraptoolFormat(rulebase_ptr);
 }
}
```

Step 2.: The **ConvertRulebaseToGraptoolFormat** will return to the **Main** function.

```
}
```



Purpose: A <u>**ConvertConditionToGraptoolFormat**</u> function will convert **condition blocks** from the LHS of the rule in the ***rule_base*** array into Graptool format. This conversion will be stored in *Graptool.rul* file.

Prototype:

```
void ConvertConditionToGraptoolFormat(char *rulebase_ptr)
{
```

A rulebase_ptr is a character pointer variable which contains the address of the first **condition block** of the rule.

Pseudo-code:

Step 1.: The computer will initialize *parenthes_count* to 1, set *condition_flag* to FALSE, and display a starting processing message.

```
/* INITIALIZE VARIABLES */
parenthes_count = 1;
condition_flag = FALSE;
Message("\n Converting condition to Graptool format...", NONE, ENDARRAY);
```

Step 2.: The computer will check the condition of the *while loop* which is a *rulebase_ptr value. If the *rulebase_ptr value is not a LHSRHS character (it is not the end of LHS rule), the computer will do step three. Otherwise, the computer will go to step five.

```
 * DO THE RULE CONDITION CONVERTING PROCESS */
while(*rulebase_ptr != LHSRHS)
```

Step 3.: The computer will first check the value of *parenthes_count*. If the **condition block** is not inside any **logical pattern block** (*parenthes_count* is TRUE), the computer will assign an **explicit and** to a **condition block** (assign LOGI_AND to *logical* variable). After that, the computer will check if the rulebase_ptr contains the beginning address of **block**. If **rulebase_ptr* is an *open parenthesis* (*rulebase_ptr contains the beginning address*), the computer will do sub-step three. Otherwise, the computer will do step four.

```
{
/* CHECK IF A CONDITION BLOCK IS NOT INSIDE ANY LOGICAL BLOCK */
if(parenthes_count == TRUE)
 logical = LOGI_AND;
/* CHECK IF IT IS THE BEGINNING OF A BLOCK */
if(*rulebase_ptr == O_PARENTHES)
```

Step 3.1.: The computer will add 1 to *parenthes_count* (because *rulebase_ptr is an open parenthesis) and assign FALSE to *good_fact* variable. The computer will then call the **GetAField** function to get the **first field** in a **block**. This first field will be stored in ***a_field*** array and its ending address will be stored in a *rulebase* character pointer variable. The computer will check if the **rulebase* value is a space. If **rulebase* is a space (this **block** has more than one **field**), the computer will search for the next *open or closed parenthesis*. The *address of the next open or closed parenthesis* will be stored in *rulebase*.

```
{
/* UPDATE A PARENTHESIS COUNTER */
parenthes_count++;
/* RE-INITIALIZE VALUE OF good_fact */
good_fact = FALSE;
/* GET THE FIRST FIELD OF A BLOCK */
rulebase = GetAField(rulebase_ptr, a_field, &notlast);
/* CHECK IF THE END OF FIRST FIELD IS A SPACE */
if(*rulebase == SPACE)
 {
 /* SEARCH FOR THE BEGINNING OF ANY SUB-BLOCK */
 while((*rulebase != O_PARENTHES) && (*rulebase != C_PARENTHES))
       rulebase++;
 }
```

Step 3.2.: The computer will first check the ***rulebase*** value (open or closed parentheses). If the ***rulebase*** is not an open parenthesis (there is no **sub-block** in the **block**), the computer will set ***good_fact*** to TRUE and assign **rulebase_ptr** value to ***rulebase***. However, if ***rulebase*** is an open parenthesis, the computer adds 1 to ***parenthes_count*** and sets ***new_logical*** to a space. The computer will then check if ***a_field*** is one of three logical operators. If ***a_field*** is "or", "and", or "not", the computer will replace the ***new_logical*** value with LOGI_OR, LOGI_AND, or LOGI_NOT characters respectively. After that, the compute will check the ***new_logical*** value. If ***new_logical*** is not a space, the computer will set ***good_fact*** to TRUE and assign the ***new_logical*** value to the ***logical*** variable.

```
/* CHECK IF THERE IS NO SUB-BLOCK */
if(*rulebase != O_PARENTHES)
 {
 good_fact = TRUE;
 rulebase = rulebase_ptr;
 }
 /* A BLOCK HAS SUB-BLOCK */
else
 {
 /* UPDATE A PARENTHESIS COUNTER */
 parenthes_count++;
 /* INITIALIZE new_logical VALUE */
 new_logical = SPACE;
 /* CHECK IF A FIRST FIELD IS LOGICAL OPERATOR OR */
 if(strcmp(a_field, "or") == SUCCESS)
   new_logical = LOGI_OR;
 else
 /* CHECK IF A FIRST FIELD IS LOGICAL OPERATOR AND */
 if(strcmp(a_field, "and") == SUCCESS)
   new_logical = LOGI_AND;
 else
 /* CHECK IF A FIRST FIELD IS LOGICAL OPERATOR NOT */
 if(strcmp(a_field, "not") == SUCCESS)
   new_logical = LOGI_NOT;
 /* IF LOGICAL OPERATOR EXISTS, UPDATE THE logical */
 /* VALUE AND SET good_fact TO TRUE */
 if(new_logical != SPACE)
  {
   good_fact = TRUE;
   logical = new_logical;
  }
 }
```

Step 3.3.: If a ***good_fact*** value is TRUE, the computer will set ***condition_flag*** to TRUE and call the **WriteFormatToWorkingFile** function to add **fact block** and its logical operator to a *Graptool.rul* file. Otherwise, the computer will search for the end of the **block**, cover the whole block with warning messages, and then ask the end user to confirm continuation of the LHS rule conversion process. If the user does not enter "Y" or "y" to confirm continuation of the process, the computer will call the **Error** function to display a message.

```
/* CHECK IF IT IS A GOOD FACT BLOCK */
if(good_fact)
 {
 condition_flag = TRUE;
 /* SET A LOGICAL OPERATOR TO THE BEGINNING OF FACT BLOCK */
 /* AND THEN WRITE THAT FACT BLOCK TO A Graptool.rul FILE */
 *(--rulebase) = logical;
 rulebase_ptr = WriteFormatToWorkingFile(CONDITION, rulebase, rptr);
 }
else
 {
 /* IF IT IS NOT A FACT BLOCK, A COMPUTER SEARCH FOR THE END */
  do
   {
    rulebase++;
    if(*rulebase == O_PARENTHES)
      parenthes_count++;
    else
    if(*rulebase == C_PARENTHES)
      parenthes_count--;
   }
  while(parenthes_count != 1);
  ch = *(++rulebase);
  *rulebase = ENDARRAY;
  /* DISPLAY A DETAIL OF A FACT BLOCK */
  Message("\n", NONE, rulebase_ptr);
  Message("\n !UNIDENTIFIED FACT BLOCK HAS BEEN ELIMINATED.!",-1,"\n");
  rulebase_ptr--;
  *rulebase_ptr = COMMENT;
  /* DO AN ELIMINATE "<-" PROCESS */
  while(*rulebase_ptr != ENDARRAY)
   {
    if((*rulebase_ptr == RE_POINT1) && (*(++rulebase_ptr) == RE_POINT2))
      *rulebase_ptr = COMMENT;
    rulebase_ptr++;
   }
  *rulebase = ch;
  rulebase_ptr = rulebase;
  /* ASK A USER TO CONFIRM PROCESS CONTINUATION */
  Message("A GRAPTOOL PROGRAM MAY NOT GIVE THE CORRECT RESULT.",-1,"\n");
  Message("ENTER Y TO CONTINUE THE PROCESS...> ",NONE, ENDARRAY);
  gets(ans);
  /* CHECK IF AN ANSWER IS NO */
  if((strcmp(ans, "Y") != SUCCESS) && (strcmp(ans, "y") != SUCCESS))
    Error("\n An user decide to not continue a program execution." ,NONE, ENDARRAY);
 }
}
```

Step 4.: The computer will update a *parenthes_count* value, increment rulebase_ptr by one, and then go back to step two.

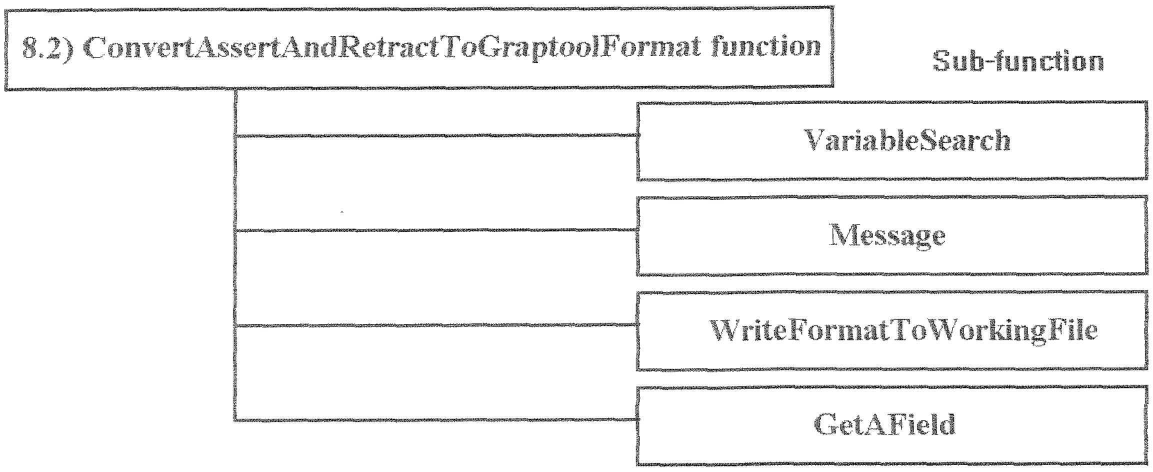
```
  /* CHECK IF IT IS A CLOSED PARENTHESIS */
  if(*rulebase_ptr == C_PARENTHES)
   parenthes_count--;
  rulebase_ptr++;
 }
```

Step 5.: The computer will check if a rule has any conditions. If a rule does not have any conditions (*condition_flag* is FALSE), the computer will set nocondition_flag to TRUE and add the special fact block to the *Graptool.rul*. This special fact block will allow the RHS of this rule to be performed one time, if and only if the *(initial-fact)* exists in the fact base.

```
/* CHECK THE EXISTENCE OF LHS RULE */
if(!condition_flag)
 {
 nocondition_flag = TRUE;
 Message("\n!!!!!!!! NO CONDITION IN THIS RULE. !!!!!!!!", NONE, ENDARRAY);
 /* INSERT THE SPECIAL FACT BLOCK TO GRAPTOOL.RUL */
 putc(CONDITION, rptr);
 fprintf(rptr, "&(initial-fact)");
 putc(CONDITION, rptr);
 fprintf(rptr, "&(pt-NoCondition-TP)");
 /* INSERT THE RETRACTION OF SPECIAL FACT BLOCK TO GRAPTOOL.RUL */
 putc(RETRACT, rptr);
 fprintf(rptr,"(pt-NoCondition-TP)");
 /* CALCULATE THE TOTAL FACT_BASE SIZE */
 fact_base_size = fact_base_size + 15;
 }
```

Step 6.: The **ConvertConditionToGraptoolFormat** function will return to the **ConvertRulebaseToGraptoolFormat** function.

```
}
```



Purpose: A <u>**ConvertAssertAndRetractToGraptoolFormat**</u> function will convert the **assert block** and **retract block** of the rule base into Graptool format.

Prototype:

```
void ConvertAssertAndRetractToGraptoolFormat(char *rulebase_ptr)
{
```

A rulebase_ptr is a character pointer variable which contains the address of the first **condition block** of rule.

Pseudo-code:

Step 1.: The computer will assign rulebase_ptr value to *rulebase*, set *assert_flag*, *retract_flag*, and *flag* to FALSE, and then search for the beginning of the RHS of the rule. The RHS beginning address will be stored in rulebase_ptr.

```
/* INITIALIZE VARIABLES VALUE */
rulebase = rulebase_ptr;
assert_flag = retract_flag = flag = FALSE;
 Message("\n Convert assert and retract to Graptool format...",NONE, ENDARRAY);
 /* SEARCH FOR THE BEGINNING OF RHS */
 rulebase_ptr = strchr(rulebase_ptr, LHSRHS);
```

Step 2.: The computer will check the condition of the *while loop* which is a *rulebase_ptr value. If *rulebase_ptr is not an ENDARRAY character (the computer dose not reach the end of ***rule_base*** array), the computer will do step three. Otherwise, the computer will go to step six.

```
 /* DO THE ACTION BLOCK CONVERSION */
 while(*rulebase_ptr != ENDARRAY)
```

Step 3.: The computer will check the value of *rulebase_ptr. If *rulebase_ptr is an open parenthesis (the beginning of an **action block**), the computer will call the <u>**GetAField**</u> function to get the **first field** of an **action block** (this first field is stored in ***a_field*** array and its ending address is stored in rulebase_ptr). The computer will then check the value of ***a_field***. If ***a_field*** matches "assert" or "retract", the computer will do the sub-step three. Otherwise, the computer will do step four. However, if *rulebase_ptr is not an open parenthesis, the computer will go to step five.

```
{
 /* CHECK IF IT IS THE BEGINNING OF AN ACTION BLOCK */
 if(*rulebase_ptr == O_PARENTHES)
  {
   /* GET THE FIRST FIELD OF AN ACTION BLOCK */
   rulebase_ptr = GetAField(rulebase_ptr, a_field, &notlast);
   /* CHECK IF AN ACTION IS ASSERT OR RETRACT BLOCK */
   if((strcmp(a_field, "assert") == SUCCESS)
                              || (strcmp(a_field, "retract") == SUCCESS))
```

Step 3.1.: The computer will initialize the variables value based on the ***a_field*** value. If the ***a_field*** value is "assert", the computer will increment the rulebase_ptr value by 1, assign ASSERT character to *flag*, and set *assert_flag* to TRUE. Otherwise, the computer will assign RETRACT character to *flag* and set *retract_flag* to TRUE.

```
    {
     /* SET THE VARIABLES FOR ASSERT BLOCK */
     if(strcmp(a_field, "assert") == SUCCESS)
      {
       rulebase_ptr++;
       flag = ASSERT;
       assert_flag = TRUE;
      }
     else
     /* SET THE VARIABLES FOR RETRACT BLOCK */
      {
       flag = RETRACT;
       retract_flag = TRUE;
      }
```

Step 3.2.: The computer will do the variable searching process by calling the **GetAField** function to get a **field of action block** (stored in ***a_field*** array). The computer will then check if ***a_field*** is a **single-field variable**. If it is, the computer will call the **VariableSearch** function to search for ***a_field*** based on the type of **action block** (***flag*** value). However; if the **action block** is an **assert block** (***flag*** is ASSERT character) and ***a_field*** is a **multifield variable**, the **VariableSearch** will also be called to search for an ***a_field***. However, if the **action block** is a **retract block** (***flag*** is RETRACT character), the computer will call **Error** function to display a warning message because every field besides the **first field in retract block** must be a **single-field variable**. The procedures in this step will continue until no more **field** is left in the **action block**.

```
    ptr = rulebase_ptr
    /* DO THE VARIABLE SEARCHING PROCESS */
    do
         {
         /* GET A FIELD FROM THE ACTION BLOCK */
         rulebase_ptr = GetAField(rulebase_ptr, a_field, &notlast);
          /*CHECK EXISTENCE OF VARIABLE FIELD IF IT IS SINGLE-FIELD VARIABLE */
          if((a_field[0] == S_WILDCARD) && (strlen(a_field) > 1))
             VariableSearch(flag, a_field, rulebase);
          else
          /* CHECK EXISTENCE OF VARIABLE FIELD IF IT CAME FROM AN
          /* ASSERT BLOCK AND IT IS A MULTIFIELD VARIABLE */
          if((flag == ASSERT) && (a_field[0] == M_WILDCARD) &&
              (a_field[1] == S_WILDCARD) && (strlen(a_field) > 2))
                VariableSearch(ASSERT, a_field, rulebase);
          else
      /* CHECK IF IT IS A RETRACT BLOCK AND NO SINGLE-FIELD VARIABLE */
          if(flag == RETRACT)
            Error("RETRACT INDEX CANNOT BE MULTIPLE WILDCARD VARIABLE.",-1,"\n");
         }
  while(notlast);
```

Step 3.3.: If the **action block** passes step 3.2 testing and the ***flag*** value is an ASSERT character, the computer will call the **WriteFormatToWorkingFile** function to write the **assert block** to the *Graptool.rul* file.

```
  /* CHECK IF ACTION BLOCK IS AN ASSERT BLOCK */
  if(flag == ASSERT)
   /* ADD ASSERT FACT STRING INTO THE GRAPTOOL.RUL FILE */
   ptr = WriteFormatToWorkingFile(ASSERT, ptr, rptr);
  }
```

Step 4.: The computer will search for the end of **action block** whose address is stored in the **rulebase_ptr** variable.

```
    /* SEARCH FOR THE ENDING OF THE ACTION BLOCK */
    rulebase_ptr = strchr(rulebase_ptr, C_PARENTHES);
   }
```

Step 5.: The computer will increment the **rulebase_ptr** value by 1 and then go back to step two.

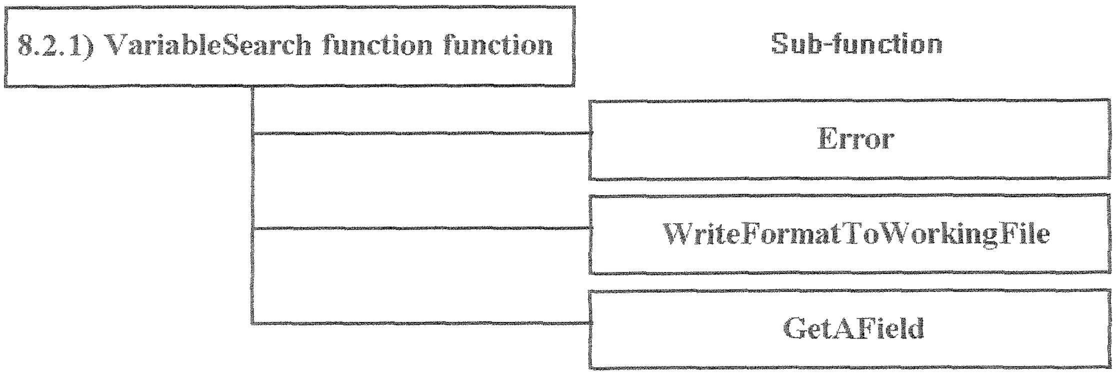
```
 rulebase_ptr++;
 }
```

Step 6.: After the computer has done the conversion process, the computer will check the ***assert_flag*** and ***retract_flag*** values. If ***assert_flag*** or ***retract_flag*** are FALSE, the computer will display warning messages. The computer will then add an ENDRF character to the *Graptool.rul* file to separate one rule from another.

```
/* CHECK IF RULE HAS ANY ASSERT BLOCK */
 if(!assert_flag)
   Message("\n!!!!!!!! NO ASSERT IN THIS RULE. !!!!!!!!!!!", NONE, ENDARRAY);
 /* CHECK IF RULE HAS ANY RETRACT BLOCK */
 if(!retract_flag)
   Message("\n!!!!!!!! NO RETRACT IN THIS RULE. !!!!!!!!!!", NONE, ENDARRAY);
 putc(ENDRF, rptr);
```

Step 7.: The **ConvertAssertAndRetractToGraptoolFormat** function will return to the **ConvertRulebaseToGraptoolFormat** function.

```
}
```



Purpose: A VariableSearch function will search the left hand side (LHS) of the rule in rule_base array for the existence of the **retract index** and **assert variable**.

Prototype:

```
void VariableSearch(int variable_type, char variable[], char *rulebase_ptr)
{
```

A variable_type is an integer variable which contains a type of **action block**. If variable_type is a RETRACT character, the computer will search for the **retract index**. However, if the variable_type is an ASSERT character, the computer will search for the **variable field**.

A variable is a character array which contains a searching variable.

A rulebase_ptr is a character pointer variable which contains the beginning address of the first rule condition.

Pseudo-code:

Step 1.: The computer will check the condition of the *while loop* which is the *rulebase_ptr value. If the *rulebase_ptr *is not a LHSRHS character (it is not the end of LHS rule), the computer will do step two.* Otherwise, the computer will go to step five.

```
/* DO THE VARIABLE SEARCHING PROCESS */
while(*rulebase_ptr != LHSRHS)
```

Step 2.: The computer will compare the first character of variable to the *rulebase_ptr value (the first character of a field from LHS). If there is a match, the computer will first decrement rulebase_ptr by 1 and then call the GetAField function to get the **field** of that first matching character (stored in ***a_field*** array). After that, the computer will do sub-step two. However, if the first character of variable does not match the *rulebase_ptr value, the computer will go to step three.

```
{
 /*CHECK IF THE FIRST CHARACTER OF variable*/
 /* MATCHES THE FIRST CHARACTER OF LHS FIELD*/
 if(variable[0] == *rulebase_ptr)
 {
   rulebase_ptr--;
   /* GET A FIELD FROM THE LEFT HAND SIDE OF THE RULE */
   rulebase_ptr = GetAField(rulebase_ptr, a_field, &notlast);
```

Step 2.1.: The computer will check the value of variable_type. If the variable_type is ASSERT character, the computer will assume that the variable is a **variable field** from **assert block**. The computer will then check for the logical field in ***a_field***. If ***a_field*** has any logical fields, the computer will delete it from the ***a_field*** array before the variable field comparison. The computer will do the comparison between ***a_field*** and variable. If they match, the computer will set the *found* variable to TRUE. Otherwise, *found* will remain FALSE. The computer will then go to step four. However, if the variable_type is not an ASSERT character, the computer will go to step 2.2.

```
  /* CHECK IF variable_type IS AN ASSERT CHARACTER */
  if(variable_type == ASSERT)
   {
    /* CHECK IF AN a_field CONTAINS ANY LOGICAL FIELD */
    if((ptr = strchr(a_field, LOGI_AND)) != ENDARRAY)
     /* ELIMINATE LOGICAL FIELD */
        *ptr = ENDARRAY;
    /* CHECK IF a_field MATCHES variable */
    if((strcmp(a_field, variable) == SUCCESS) && (parenthes_count > 1))
        found = TRUE;
   }
```

Step 2.2.: The computer will check the variable_type value. If the variable type is a RETRACT character, the computer will look for a **retract index** (a **field** which is followed by <-) that matches the variable. If the ***a_field*** is that **retract index**, the computer will set *found* to TRUE and call the **WriteFormatToWorkingFile** function to add a **removing block** to the *Graptool.rul* file. After that, the computer will go to step four. However, if the variable_type is not a RETRACT character, the computer will also go to step four.

```
    else
    /* CHECK IF variable_type IS A RETRACT CHARACTER */
    if(variable_type == RETRACT)
     {
      /* CHECK FOR THE EXISTENCE OF variable */
      ptr = rulebase_ptr;
      if((strcmp(a_field, variable) == SUCCESS) && (*(++ptr) == RE_POINT1)
                              && (*(++ptr) == RE_POINT2) && (*(++ptr) != COMMENT))
       {
        found = TRUE;
        parenthes_count++;
        /* SEARCH FOR THE END OF RETRACT FACT */
        rulebase_ptr = strchr(rulebase_ptr, O_PARENTHES);
        /* ADD RETRACT FACT TO THE Graptool.rul FILE */
        rulebase_ptr = WriteFormatToWorkingFile(RETRACT, rulebase_ptr, rptr);
       }
     }
   }
```

Step 3.: The computer will perform this step if the first character of variable does not match the *rulebase_ptr value. The computer will then check if the *rulebase_ptr is a quotation mark. If it is TRUE, the computer will search for the other quotation mark and that quote address will be saved in rulebase_ptr.

```
  else
  if(*rulebase_ptr == QUOTE)
   rulebase_ptr = strchr(++rulebase_ptr, QUOTE);
```

Step 4.: The computer will update the *parenthes_count* value by adding 1 to *parenthes_count* if *rulebase_ptr is an open parenthesis, or subtracting 1 from *parenthes_count* if *rulebase_ptr is a closed parenthesis. The computer will also increment rulebase_ptr by 1, and then go back to step one.

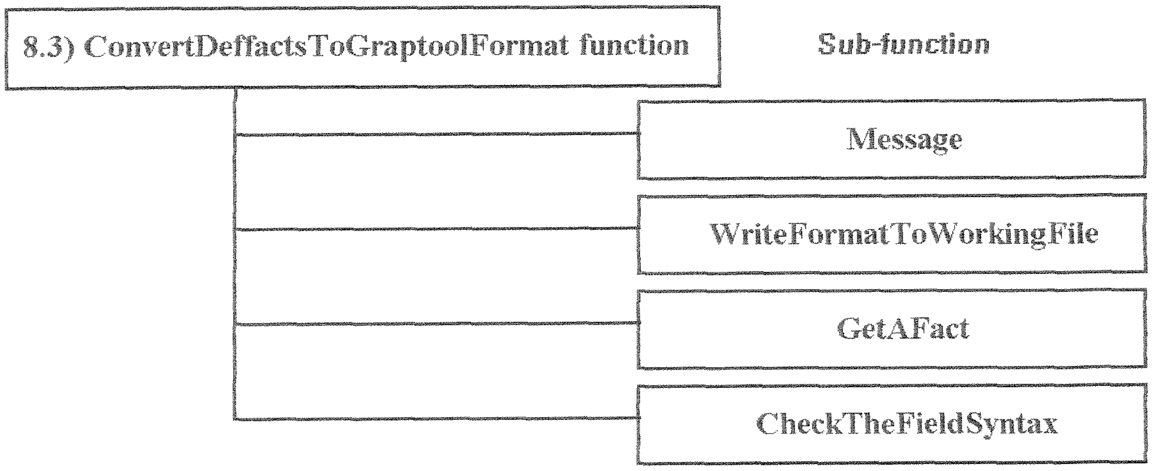
```
  if(*rulebase_ptr == O_PARENTHES)
   parenthes_count++;
  else
  if(*rulebase_ptr == C_PARENTHES)
   parenthes_count--;
  rulebase_ptr++;
  }
```

Step 5.: If the *found* value is FALSE, the computer will display an error message.

```
if(!found)
  Error(variable, NONE, " CANNOT BE FOUND IN THIS RULE. \n");
```

Step 6.: The **VariableSearch** function will return to the **ConvertAssertAndRetractToGraptoolFormat** function.

```
}
```



Purpose: A <u>**ConvertDeffactsToGraptoolFormat**</u> function will convert the **deffacts construct** in the ***rule_base*** array into Graptool format. The result of this function will be saved in the *Graptool.fac* file.

Prototype:

```
void ConvertDeffactsToGraptoolFormat(char *rulebase_ptr)
{
```

A rulebase_ptr is a character pointer variable which contains the beginning address of the first **fact block** in the ***rule_base*** array.

Pseudo-code:

Step 1.: The computer will set *parenthes_count* to 1, set *deffacts_flag* to FALSE, and display a process message.

```
 /* INITIALIZE THE VARIABLES */
parenthes_count = 1;
deffacts_flag = FALSE;
Message("\n Convert deffacts to Graptool format........",NONE, ENDARRAY);
```

Step 2.: This is the beginning of the deffacts construct converting process. The computer will begin by checking the *while loop* condition which is a rulebase_ptr value. If the *rulebase_ptr is not an ENDARRAY character, the computer will search for the beginning of the **fact block** (an open parenthesis) and do the next step of the procedure. Otherwise, the computer will go to step four.

```
 /* DO THE DEFFACTS CONSTRUCT CONVERTING PROCESS */
while(*rulebase_ptr != ENDARRAY)
 {
 rulebase_ptr = strchr(rulebase_ptr, O_PARENTHES);
```

Step 3.: If the computer finds an open parenthesis, (*rulebase_ptr is an open parenthesis) it will do sub-step three. Otherwise, the computer will go back to step two.

```
 /* CHECK IF IT IS THE BEGINNING OF FACT */
if(*rulebase_ptr == O_PARENTHES)
```

Step 3.1.: The computer will assign zero to *i*, increment *parenthes_count* by 1, set the first character of ***a_fact*** array to an open parenthesis, and assign the rulebase_ptr value to *fact_ptr*.

```
 {
  /* INITIALIZE THE VARIABLE */
  i = 0;
  parenthes_count++;
  fact_ptr = rulebase_ptr;
  a_fact[0] = O_PARENTHES;
```

Step 3.2.: The computer will read a **block** from the ***rule_base*** array to ***a_fact***. Any time the computer reads a closed parenthesis, *parenthes_count* will be subtracted by 1. If the computer finds an open parenthesis, *parenthes_count* will be added by 1. The procedures in this step will continue until the computer finds a closed parenthesis (*rulebase_ptr is a closed parenthesis).

```
   /* READ A FACT FROM rule_base ARRAY INTO a_fact ARRAY */
  do
    {
     +;
     rulebase_ptr++;
     /* SUBTRACT ONE IF IT IS A CLOSED PARENTHESIS */
     if(*rulebase_ptr == C_PARENTHES)
       parenthes_count--;
     else
     /* ADD ONE IF IT IS AN OPEN PARENTHESIS */
     if(*rulebase_ptr == O_PARENTHES)
       parenthes_count++;
      /* ADD FACT INTO a_fact ARRAY */
     a_fact[i] = *rulebase_ptr;
    }
  while(*rulebase_ptr != C_PARENTHES);
 a_fact[++i] = ENDARRAY;
```

Step 3.3.: The computer will check the value of *parenthes_count*. If the *parenthes_count* value is TRUE (one), meaning that there is no **sub-block** in the block being read (***a_fact*** array), the computer will do step 3.4. Otherwise, the computer will do step 3.5.

```
  /* CHECK IF FACT BEING READ HAS ANY SUB-BLOCK */
  if(parenthes_count == TRUE)
```

Step 3.4.: The computer will call **CheckTheFieldSyntax** function to check the syntax of each field in ***a_fact***. If the **CheckTheFieldSyntax** does not find any error (*good_fact* is TRUE), the computer will set *fact_flag* and *deffacts_flag* to TRUE, and then call the **WriteFormatToWorkingFile** function to write the ***a_fact*** array into the *Graptool.fac* file. However, if the **CheckTheFieldSyntax** finds an error (*good_fact* is FALSE), the computer will display a warning message. After the computer has done step 3.4, it will go back to step two.

```
 {
 /* CHECK THE FIELD SYNTAX ERROR */
  good_fact = CheckTheFieldSyntax(a_fact);
  /* CHECK IF IT IS A GOOD FACT */
  if(good_fact)
   {
    fact_flag = deffacts_flag = TRUE;
    /* ADD A GOOD FACT TO THE Graptool.fac FILE */
    WriteFormatToWorkingFile(CONDITION, a_fact, cptr);
   }
  else
   {
    Message("\n", NONE, a_fact);
    Message("\n Warning THE ABOVE FACT HAS SYNTAX ERROR.",-1, ENDARRAY);
   }
 }
```

Step 3.5 .: The computer will search for the end of the block being read, which may contain many **sub-blocks**. It will then display that block with a warning message. The computer will go back to step two in which **rulebase_ptr** contains the ending address of the block being read.

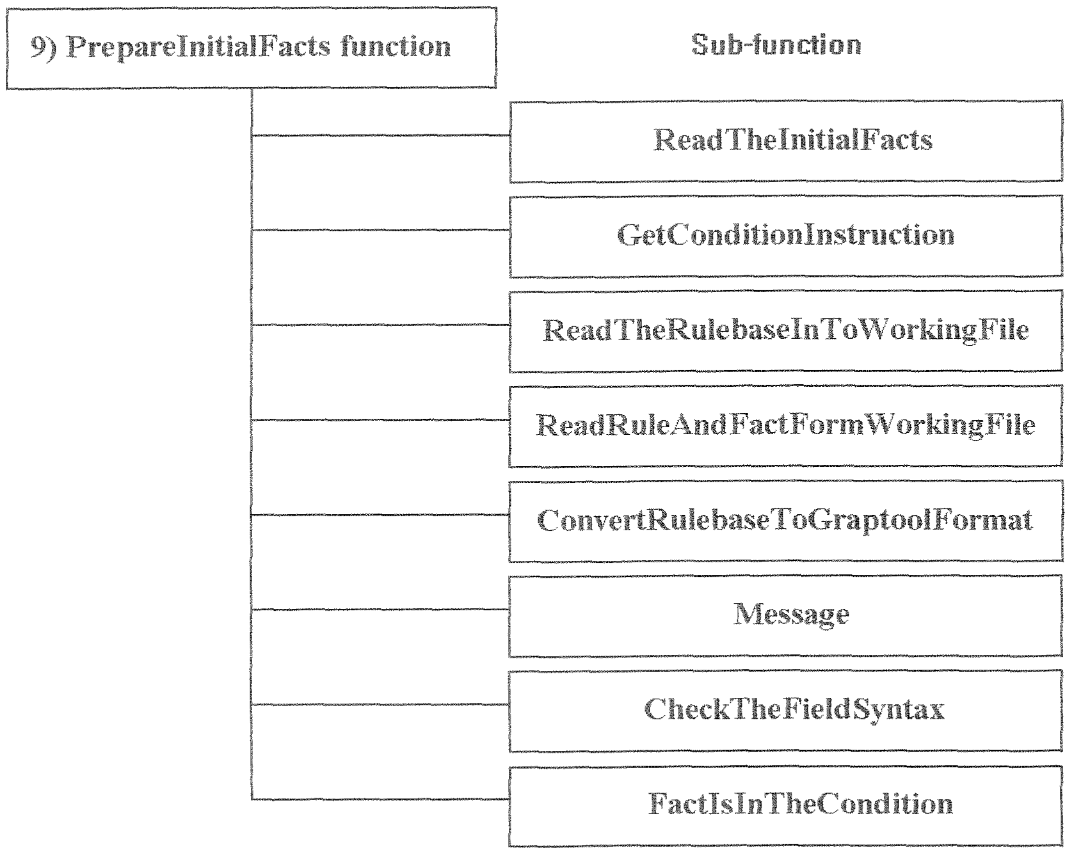
```
 else
  {
   /* SEARCH FOR THE END OF THE BLOCK BEING READ */
   while(parenthes_count != TRUE)
     {
      rulebase_ptr++;
      if(*rulebase_ptr == O_PARENTHES)
            parenthes_count++;
      else
      if(*rulebase_ptr == C_PARENTHES)
            parenthes_count--;
     }
   ch = *(++rulebase_ptr);
   *rulebase_ptr = ENDARRAY;
   Message("\n", NONE, fact_ptr);
   Message("\n THE UNIDENTIFIED FACT HAS BEEN ELIMINATED.", NONE, ENDARRAY);
   *rulebase_ptr = ch;
  }
 }
}
```

Step 4.: The computer will check for the existence of any **fact block** by checking the value of *deffacts_flag*. If *deffacts_flag* is not TRUE (the computer does not find any **fact block**), the computer will display a warning message. Otherwise, the computer will **add** an ENDRF character to the *Graptool.fac* file.

```
/* CHECK IF IT HAS NO INITIAL FACT */
if(!deffacts_flag)
  Message("\n!!!!!! NO GOOD INITIAL FACT IN THIS DEFFACTS. !!!!!!", NONE, "\n");
else
  putc(ENDRF, cptr);
```

Step 5.: The **ConvertDeffactsToGraptoolFormat** function will return to the **ConvertRulebaseToGraptoolFormat** function.

```
}
```


9) PrepareInitialFacts function
Sub-function
ReadTheInitialFacts
GetConditionInstruction
ReadTheRulebaseInToWorkingFile
ReadRuleAndFactFormWorkingFile
ConvertRulebaseToGraptoolFormat
Message
CheckTheFieldSyntax
FactIsInTheCondition

Purpose: A <u>**PrepareInitialFacts**</u> function will prepare the initial facts (**fact blocks**) for the rule-based structure testing process. If initial facts already exist in the *Graptool.fac* file, the computer will read that initial facts into the ***fact_base*** array. The end user may also add extra facts by keyboard. However, if initial facts do not exist, the end user has two choices: to enter the name of the file which has the initial facts, or to enter all the facts by hand.

Prototype:

```
void PrepareInitialFacts(void)
{
```

There is no prototype in this function.

Pseudo-code:

Step 1.: The computer will check for the existence of initial facts. If the initial fact does not exist (fact_flag is FALSE), the computer will do sub-step one. Otherwise the computer will go to step two.

```
/* CHECK IF INITIAL FACTS ALREADY EXIST */
if(!fact_flag)
```

Step 1.1.: The computer will ask the end user to either press enter (select to enter all initial facts by hand) or to enter the name of a file which has the initial facts. The end user's answer will be stored in the ***deffacts_file*** array.

```
{
printf("\n!!!!!!!!!! THE INITIAL FACT DOES NOT EXIST !!!!!!!!!");
printf("\n ! If you want to enter facts by hand, press enter   !");
printf("\n ! otherwise enter the deffacts file name............!\n");
printf("\n PATH AND FILE NAME CANNOT BE LONGER THAN %d BYTES." ,PATH_SIZE);
printf("\n Enter the deffacts file name -> ");
/* GET THE END USER SELECTION */
gets(deffacts_file);
```

Step 1.2.: The computer will open the ***deffacts_file*** file. If the file is successfully opened, the **ReadTheRulebaseInToWorkingFile**, the **ReadRuleAndFactFromWorkingFile**, and the **ConvertRulebaseToGraptoolFormat** functions will be called to perform error checking and to convert **deffacts construct** into Graptool format. The **fact_flag** will be set to TRUE by the **ConvertRulebaseToGraptoolFormat** function and the computer will then go to step two. However, if the computer cannot open the file for any reason (**fopen(deffacts_file, "r")** returns null), the computer will go to step two.

```
/* CHECK IF  THE SELECTED FILE CAN BE OPENED */
if((fptr = fopen(deffacts_file, "r")) != NULL)
 {
 Message("\n@@@@ THE FACT FILE IS ", NONE, deffacts_file);
 iptr = fopen(iname, "wt+");
 ReadTheRulebaseInToWorkingFile();
 /* DO GRAPTOOL FORMAT CONVERSION PROCESS */
 while(!feof(iptr))
  {
  done = ReadRuleAndFactFromWorkingFile(DEFFACTS);
  /* CHECK IF IT IS A DEFFACTS CONSTRUCT */
  if(done == DEFFACTS)
    ConvertRulebaseToGraptoolFormat(done);
  }
 }
}
```

Step 2.: The computer will prepare the ***fact_base*** array for its initialization by checking the value of **nocondition_flag**. If **nocondition_flag** is TRUE (some rule has no condition), the computer will copy a special **fact block** "(pt-NoCondition-TP)" to the ***fact_base***. Otherwise, the computer will add an ENDARRAY character to the beginning of ***fact_base*** (clear all the values in the array).

```
/* CHECK IF ANY OF THE RULES HAVE NO CONDITION */
if(nocondition_flag)
  /* COPY A SPECIAL FACT BLOCK TO FACT BASE */
  strcpy(fact_base, "(pt-NoCondition-TP)");
else
  /* CLEAR ALL THE VALUES IN THE FACT BASE */
  fact_base[0] = ENDARRAY;
```

Step 3.: The computer will check the fact_flag value. If fact_flag is TRUE, the computer will call the **ReadTheInitialFacts** function to read all the initial facts from the *Graptool.fac* file into the ***fact_base***. It will also store the ending address of ***fact_base*** to *con_ptr* variable. The computer will allow the user to enter extra facts by entering "Y" or "y" which is stored in ***ans*** array. However, if fact_flag is still FALSE, the computer will copy Y to the ***ans***, get the current ending address of ***fact_base*** (store in *con_ptr* variable), and display a warning message.

```
/* CHECK IF INITIAL FACTS ALREADY EXIST *
 if(fact_flag)
 {
 /* READ INITIAL FACTS INTO THE FACT BASE */
 con_ptr = ReadTheInitialFacts();
 printf("\n Do you want to add the extra facts to fact base(N)? ");
 gets(ans);
 }
/* THE INITIAL FACT DOES NOT EXIST */
else
 {
 strcpy(ans, "Y");
  /* GET THE ENDING ADDRESS OF fact_base ARRAY */
 con_ptr = strlen(fact_base);
 Message("\n\n YOU WILL ENTER ALL THE INITIAL FACTS BY HAND." ,NONE, "\n");
 }
```

Step 4.: This step is the beginning of entering initial facts by hand. The purpose of this loop is to give the end user a second chance to re-enter all initial facts (previous entering of initial facts will be erased). The computer will first check the *while loop* condition which is the ***ans*** value. If ***ans*** is "Y" or "y", the computer will assign *con_ptr* value to the *i* variable, assign TRUE to the *notdone* variable, set *limit* value to CON_SIZE minus two, and assign an ENDARRAY character to a ***fact_base*** array address *i* (all values after the *i* address will be deleted). Then, the computer will call the **GetConditionInstruction** function to display the rule and instructions for entering facts, and do sub-step four. However, if ***ans*** does not equal "Y" or "y", the computer will go to step five.

```
/* DO THE ENTERING INITIAL FACTS BY HAND PROCESS */
while((strcmp(ANS, "Y") == SUCCESS) || (strcmp(ans, "y") == SUCCESS))
 {
 /* INITIALIZE THE PROCESSING VARIABLES */
 i = con_ptr;
 notdone = TRUE;
 limit = CON_SIZE - 2;
 fact_base[i] = ENDARRAY;
 GetConditionInstruction();
```

Step 4.1.: This step is the beginning of the other *while loop*. This loop will continue until the *notdone* value is FALSE. When the end user has finished or decided to stop entering the initial facts, he will not enter an open parenthesis after a "Enter the initial fact......." message. The computer will go to step 4.4 after it gets out from the *while loop*. However, if *notdone* is TRUE, the computer will set *fact_len* to FACT_SIZE minus two, initialize *parenthes_count*, *fact_ptr*, *quote_count*, and *result.rem* to SUCCESS, and then go to step 4.2 to do the getting initial fact process.

```
/* DO THE GETTING AND CONVERTING INITIAL FACT PROCESS */
while(notdone)
 {
 /* INITIALIZE THE PROCESSING VARIABLES */
 fact_len = FACT_SIZE - 2;
 parenthes_count = fact-ptr = quote_count = result.rem = SUCCESS;
 printf("\n!! FACT STRING CANNOT BE LONGER THAN %d BYTES. !!",FACT_SIZE);
 printf("\n!! EACH FIELD CANNOT BE LONGER THAN %d BYTES.  !!",FIELD_SIZE);
 printf("\n** You have %d bytes left in the fact_base array. **",limit-i);
 printf("\n Enter the fact string ........");
 printf("\n12345678901234567890123456789012345678901234567890123456789 0");
 printf("1234567890123456789\n");
```

Step 4.2.: The computer will read the initial facts one character at a time from the keyboard (stored it in the *ch* variable), and check for errors at the same time. Like the processes in the **ReadTheRulebaseInToWorkingFile** function and the **ReadRuleAndFactFromWorkingFile** function, the computer will eliminate a space after some characters, add a space in front of others, ignore some characters, etc. If the computer accepts the entered character (*acceptable_letter* is TRUE), it will add *ch* value to the ***fact*** array and display it on the screen. This process will continue until the end user presses enter (↵), or the ***fact*** array size is in excess.

```
 /* GET AN INITIAL FACT FROM THE KEYBOARD */
 while((fact_ptr < fact_len) && ((ch = getch()) != '\r'))
  {
  acceptable_letter = TRUE;
  /* ENTER THE UNPRINTABLE CHARACTER */
  if(!isprint(ch))
    acceptable_letter = FALSE;
  else
   /* CHECK IF ch IS A SPACE */
   if(ch == SPACE)
   {
    /* ELIMINATE SPACES AFTER SPACE */
    if(fact[fact_ptr-1] == SPACE)
      acceptable_letter = FALSE;
    else
    /* ELIMINATE SPACES AFTER FIRST QUOTE */
    if((fact[fact_ptr-1] == QUOTE) && (result.rem == TRUE))
      acceptable_letter = FALSE;
    else
    /* ELIMINATE SPACES AFTER OPEN PARENTHESIS OUTSIDE THE QUOTE */
    if((fact[fact_ptr-1] == O_PARENTHES) && (result.rem == SUCCESS))
      acceptable_letter = FALSE;
  }
  else
  /* CHECK IF ch IS A QUOTE */
  if(ch == QUOTE)
   {
    /* CALCULATE THE QUOTE AND ch POSITION */
    result = div((++quote_count), 2);
      /* ELIMINATE SPACE AFTER THE SECOND QUOTE */
    if((result.rem == SUCCESS) && (fact[fact_ptr-1] == SPACE))
       fact_ptr--;
    else
    /* ADD A SPACE IN FRONT OF THE FIRST QUOTE IF IT DOSE */
    /* NOT FOLLOW AN OPEN PARENTHESIS AND A SPACE */
    if((result.rem == TRUE) &&
       (fact[fact_ptr-1] != SPACE) && (fact[fact_ptr-1] != O_PARENTHES))
     {
      fact[fact_ptr] = SPACE;
      printf("%c", SPACE);
      fact_ptr++;
     }
   }
  else
  /* ELIMINATE A SPACE AFTER THE CLOSED PARENTHESIS OUTSIDE THE QUOTE */
  if((ch == C_PARENTHES) &&
          (result.rem == SUCCESS) && (fact[fact_ptr-1] == SPACE))
    fact_ptr--;
  else
  /* ADD A SPACE AFTER THE SECOND QUOTE IF IT IS */
   /* NOT FOLLOWED BY A CLOSED PARENTHESIS AND A SPACE*/
  if((ch != SPACE) && (ch != C_PARENTHES) &&
          (result.rem == SUCCESS) && (fact[fact_ptr-1] == QUOTE))
   {
    fact[fact_ptr] = SPACE;
    printf("%c", SPACE);
    fact_ptr++;
   }
 /* UPDATE THE PARENTHESIS COUNTER */
 if(((ch == O_PARENTHES) ||
                 (ch == C_PARENTHES)) && (result.rem == SUCCESS))
   parenthes_count++;
  /* CHECK IF ch IS AN ACCEPTABLE CHARACTER */
  if(acceptable_letter)
   {
    /* ADD ch TO THE fact ARRAY AND DISPLAY IT ON THE SCREEN */
    fact[fact_ptr] = (char)ch;
    printf("%c", ch);
    fact_ptr++;
    }
  }
fact[fact_ptr] = ENDARRAY;
```

Step 4.3.: After the computer finishes getting initial fact from the keyboard process, it will check the first character in the ***fact*** array. If the first character is not an open parenthesis, the computer will assume that the user wants to stop entering initial facts. The computer will then assign FALSE to *notdone* and go back to step 4.1. Otherwise, it will do sub-step 4.3.

```
/* CHECK IF THE FIRST CHARACTER IN fact IS AN OPEN PARENTHESIS */
if(fact[0] != O_PARENTHES)
  notdone = FALSE;
```

Step 4.3.1.: The computer will check for general errors in a **fact block** such as missing quotation marks, missing closed parentheses, etc. If it detects any errors, the error message will be displayed and the computer will go back to step 4.1.

```
else
 {
  Message("\n Processing initial fact is.......\n" ,NONE, fact);
  fact_len = strlen(fact) - 1;
  /* CHECK IF THE INITIAL FACT IS MISSING A QUOTE */
  if(result.rem != SUCCESS)
     Message("\n ERROR.. MISSING QUOTE IN THE INITIAL FACT." ,NONE, "\n");
  else
  /* CHECK IF THE INITIAL FACT IS MISSING CLOSED PARENTHESIS */
  if(fact[fact_len] != C_PARENTHES)
     Message("\n ERROR.. MISSING CLOSED PARENTHESIS AT THE END." ,NONE, "\n");
  else
  /* CHECK IF INITIAL FACT HAS A SUB-FACT */
  if(parenthes_count > 2)
     Message("\n ERROR..!!!! UNIDENTIFIED INITIAL FACT !!!!",NONE, "\n");
  else
  /* CHECK THE INITIAL FACT SIZE */
  if((i + fact_len) >= limit)
    Message("\n ERROR.. FACT IS TOO LONG FOR THE FACT_BASE ARRAY.",-1,"\n");
```

Step 4.3.2.: The computer will replace all spaces, open parentheses, and closed parentheses between quotes with Q_SPACE, O_PAREN, and C_PAREN characters. It will then calculate the size of the biggest field and store that calculation in *field_size* in the *fact* array. If that size exceeds the initialize field size (*field_size* is greater than FIELD_SIZE minus two), the computer will display an error message and go back to step 4.1.

```
else
  {
   quote_count = result.rem = SUCCESS;
   /* DO THE REPLACING CHARACTER PROCESS */
    for(fact_ptr = 0; fact_ptr <= fact_len; fact_ptr++)
        {
      /* CHECK IF IT IS A QUOTE */
          if(fact[fact_ptr] == QUOTE)
        /* UPDATE THE CHARACTER POSITION */
            result = div((++quote_count), 2);
      /* CHECK IF A CHARACTER IS BETWEEN QUOTES */
          if(result.rem == TRUE)
          {
           /* REPLACE SPACES WITH Q_SPACE CHARACTER */
           if(fact[fact_ptr] == SPACE)
            fact[fact_ptr] = Q_SPACE;
           else
            /* REPLACE OPEN PARENTHESIS WITH O_PAREN CHARACTER */
            if(fact[fact_ptr] == O_PARENTHES)
              fact[fact_ptr] = O_PAREN;
            else
             /* REPLACE CLOSED PARENTHESIS WITH C_PAREN CHARACTER */
             if(fact[fact_ptr] == C_PARENTHES)
                fact[fact_ptr] = C_PAREN;
          }
        }
    space_count = field_start = field_size = result.rem = SUCCESS;
    /* CALCULATE THE MAXIMUM FIELD SIZE OF THE GIVEN FACT */
    for(fact_ptr = 0; fact_ptr <= fact_len; fact_ptr++)
     {
          if(ch == SPACE)
           {
            result = div((++space_count), 2);
            if(result.rem == TRUE)
              field_start = fact_ptr;
            else
            if(field_size < (fact_ptr - field_start + 1))
              field_size = fact_ptr - field_start + 1;
          }
    }
   if(field_size == SUCCESS)
        field_size = fact_len + 1;
   /* CHECK IF IT EXCEEDS AN INITIALIZE FIELD SIZE */
   if(field_size >= FIELD_SIZE - 2)
        Message("\n ERROR.. FIELD IS TOO BIG FOR THE FIELD ARRAY.",-1,"\n");
```

Step 4.3.3.: The computer will call **CheckTheFieldSyntax** function to check the **fields** syntax in ***fact*** array. If the **CheckTheFieldSyntax** finds any error field (*good_fact* is FALSE), the computer will display an error message and go back to step 4.1.

```
    else
     {
          /* CHECK FOR THE FIELD SYNTAX ERROR */
          good_fact = CheckTheFieldSyntax(fact);
      /* CHECK IF IT IS AN ERROR FIELD */
           if(!good_fact)
             Message("\n ERROR.. GIVEN FACT HAS SYNTAX ERROR.", NONE, "\n");
```

Step 4.4.4.: The computer will call **FactIsInTheFactbase** function to check for duplicated initial facts (from ***fact*** array) in the ***fact_base*** array. If the **FactIsInTheFactbase** finds that initial fact (*good_fact* is TRUE), the computer will display an error message and go back to step 4.1. Otherwise, the computer will add this initial fact to the ***fact_base*** and then go back to step 4.1.

```
            else
             {
              /* CHECK FOR THE EXISTENCE OF INITIAL FACT IN THE FACT BASE */
              FactIsInTheFactbase(fact, &good_fact);
        /* CHECK IF FIELD EXISTS IN THE FACT BASE */
              if(good_fact)
                Message("\n", NONE, "ALREADY EXISTS. <!!>\n");
              else
               {
                /* ADD THE INITIAL FACT INTO THE fact_base ARRAY */
                Message("\n", NONE, "HAS BEEN ACCEPTED. <**>\n");
                strcat(fact_base, fact);
                i = strlen(fact_base);
               }
             }
     }
    }
   }
  }
```

Step 4.4.: The computer will ask the end user to confirm the accuracy of the initial facts entered by keyboard. If the end user wishes to change any of the initial facts, he must re-enter all the initial facts again. The user can delete all entered initial facts by simply answering "Y" or "y" to the question. The computer will go back to step four.

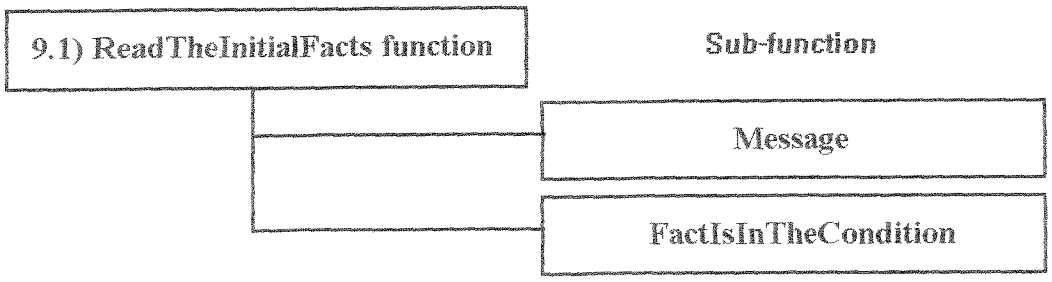
```
printf("\n\n @@@ ALL INPUT FACTS WILL BE DELETED IF YOU SELECT 'Y.' @@@\n");
printf("Do you want to re-enter the facts in fact base(N)? ");
gets(ans);
}
```

Step 5.: The computer will consider the ***fact_base*** to have initial facts if its size is greater than zero and it contains any **fact blocks** excluding the "(pt-NoCondition-PT)". The computer will then assign TRUE to the fact_flag variable. Otherwise, the fact_flag remains FALSE.

```
if((strlen(fact_base) > 0)
        && (strcmp(fact_base,"(pt-NoCondition-TP)") != SUCCESS))
 fact_flag = TRUE;
```

Step 6.: The **PrepareInitialFacts** will return to the Main function.

```
}
```



Purpose: A **<u>ReadTheInitialFacts</u>** function will read the **fact blocks** (initial facts) from the *Graptool.fac* file into the ***fact_base*** array.

Prototype:

```
int ReadTheInitialFacts(void)
{
```

There is no prototype in this function.

Pseudo-code:

Step 1.: The computer will display a processing message and set the *Graptool.fac* file pointer to the beginning of the file. The computer will find the current ending address of ***fact_base*** array and store it in ***con_ptr*** variable.

```
Message("\n<<<< Start reading facts from Graptool.fac >>>>\n",-1, ENDARRAY);
/* SET Graptool.fac FILE POINTER TO THE BEGINNING */
rewind(cptr);
/* GET A CURRENT ENDING OF fact_base ARRAY */
con_ptr = strlen(fact_base);
```

Step 2.: The computer will start the initial fact reading process by checking the condition of the *while loop* which is the returning value of **<u>feof(cptr)</u>**. If the end of *Graptool.fac* has been reached (**<u>feof(cptr)</u>** returns TRUE), the computer will do step three. Otherwise, the computer will do sub-step two.

```
/* DO THE INITIAL FACT READING PROCESS */
while(!feof(cptr))
```

Step 2.1.: The computer will read a character from the *Graptool.fac* and this character will be stored in a ***ch*** variable. If ***ch*** is a DEFFACTS character (indicate the beginning of a initial fact set), the computer will do step 2.2. Otherwise, the computer will go back to step two.

```
{
ch = getc(cptr);
/* CHECK IF IT IS THE BEGINNING OF INITIAL FACT SET */
if(ch == DEFFACTS)
```

Step 2.2.: The computer will first get the name of the initial fact set being read, and store it in the ***deffacts_name*** array. The computer will then read the initial facts from the *Graptool.fac* file into the ***fact_base*** array. After that, the computer will display the name of the initial fact set and go back to step two.

```
  {
   i = 0;
   ch = getc(cptr);
   /* GET THE NAME OF THE INITIAL FACT SET */
   while((ch = getc(cptr)) != C_PARENTHES)
    {
     deffacts_name[i] = ch;
      +;
    }
   deffacts_name[i] = ENDARRAY;
   /* DO THE READ INITIAL FACT INTO fact_base ARRAY */
   while(((ch = getc(cptr)) != ENDRF) && (!feof(cptr)))
    {
     if(ch != CONDITION)
      {
       /* READ THE INITIAL FACTS INTO THE FACT_BASE ARRAY */
       fact_base[con_ptr] = ch;
       con_ptr++;
      }
    }
   /* CHECK IF IT IS THE END OF INITIAL FACT SET */
   if(ch == ENDRF)
    /* DISPLAY THE NAME OF INITIAL FACT SET */
    Message("\n FINISH READING DEFFACTS...", NONE, deffacts_name);
  }
 }
fact_base[con_ptr] = ENDARRAY;
```

Step 3.: The computer will close the *Graptool.fac* and call the **FactIsInTheFactbase** function to check for the existence of "(initial-fact)" in the ***fact_base*** array. The result of **FactIsInTheFactbase** will be stored in the *found* variable.

```
/* CLOSE THE Graptool.fac FILE */
fclose(cptr);
FactIsInTheFactbase("(initial-fact)", &found);
```

Step 4.: If "(initial-fact) is not found in the ***fact_base*** (*found* is FALSE), the computer will add the "(initial-fact)" to the end of the ***fact_base*** array.

```
/* CHECK IF (initial-fact) IS NOT IN fact_base ARRAY */
if(!found)
  /* ADD (initial-fact) TO THE END OF fact_base ARRAY */
  strcat(fact_base,"(initial-fact)");
```

Step 5.: The computer will display the ending process message, and re-calculate the ending address of ***fact_base*** array.

```
Message("\n\n<<<< End reading facts from the Graptool.fac >>>>\n",-1,"\0");
/* RE-CALCULATE THE END ADDRESS OF fact_base ARRAY */
con_ptr = strlen(fact_base);
```

Step 6.: The **ReadTheInitialFacts** will return to the **PrepareInitialFacts** function with *con_ptr* value.

```
return(con_ptr);
}
```

9.2) GetConditionInstruction function

Purpose: A **GetConditionInstruction** function will provide instructions on how to enter initial facts by keyboard to the Graptool software.

Pseudo-code:

```
void GetConditionInstruction(void)
{
printf("\n***************** IMPORTANT INSTRUCTIONS ********************");
printf("\n The size of the fact_base array is %d bytes. ",CON_SIZE);
printf("\n EVERY INITIAL FACT MUST START AND END WITH A PARENTHESES.");
printf("\n Otherwise the computer will not accept the given fact.");
printf("\n PRESS RETURN(ENTER) AFTER FINISHING INPUT INTO INITIAL FACT.");
printf("\n Otherwise the computer will not start the fact process.");
printf("\n The computer will accept ONLY ONE FACT AT A TIME.");
printf("\n The syntax of the fact is the SAME as the fact of CLIPS.");
printf("\n***************************************************************\n");
}
```

10) ReadRuelbaseFromWorkFile function

Purpose: A **ReadRulebaseFromWorkingFile** function will read the rule bases from the *Graptool.rul* file into the ***rule_base*** array.

Prototype:

```
void ReadRulebaseFromWorkingFile(void)
{
```

There is no prototype in this function.

Pseudo-code:

Step 1.: The computer will set the *Graptool.rul* file and the information file pointers to the beginning of each file.

```
/* SET Graptool.rul FILE POINTER TO THE BEGINNING */
rewind(rptr);
/* SET INFORMATION FILE POINTER TO THE BEGINNING */
rewind(iptr);
printf("\n Rule number 0 is the initial-state. \n");
fprintf(iptr, "Rule number 0 is the initial-state. \n");
```

Step 2.: The computer will start the rule base reading process by checking the condition of the *while loop* (which is the returned value of **feof(rptr)**). If the end of *Graptool.rul* has been reached (**feof(rptr)** returns TRUE), the computer will do step three. Otherwise, the computer will do sub-step two.

```
/* DO THE RULE BASE READING PROCESS */
while(!feof(rptr))
```

Step 2.1.: The computer will read a character from the *Graptool.rul* and store it in a *ch* variable. If *ch* is a DEFRULE character (indicating the beginning of a rule base), the computer will do step 2.2. Otherwise, the computer will go back to step two.

```
{
ch = getc(rptr);
/*CHECK IF IT IS THE BEGINNING OF RULE */
if(ch == DEFRULE)
```

Step 2.2.: The computer will read the next character from the *Graptool.rul* (stored in the ch variable) and display a number with a name of the rule being read on the screen. It will also save that display in the information file.

```
{
ch = getc(rptr);
/* DISPLAY THE NUMBER OF RULES */
printf("Rule number %d is the ", rule_counter);
fprintf(iptr ,"Rule number %d is the ", rule_counter);
 /* GET AND DISPLAY RULE BASE NAME */
while((ch = getc(rptr)) != C_PARENTHES)
 {
  printf("%c", ch);
  fprintf(iptr, "%c" ,ch);
 }
rule_base[i] = DEFRULE;
printf(".\n");
fprintf(iptr, "\n");
```

Step 2.3.: The computer will read a rule base from the *Graptool.rul* to the ***rule_base*** array. After the computer finishes the reading process, the computer will increment the value of rule_counter (rule number) by 1 and then go back to step two.

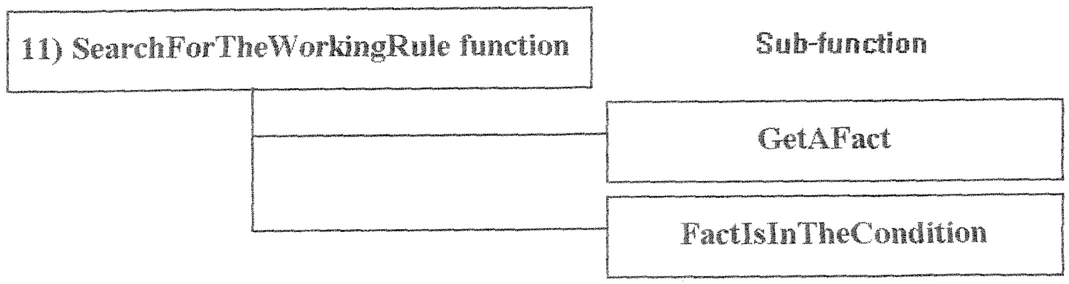
```
+;
/* DO THE READING RULE PROCESS */
while((ch = getc(rptr)) != ENDRF)
 {
  /* WRITE A RULE INTO THE rule_base ARRAY */
  rule_base[i] = ch;
  +;
 }
rule_base[i] = ENDRF;
rule_base[++i] = ENDARRAY;
 /* INCREMENT THE RULE NUMBER BY ONE */
rule_counter++;
}
}
```

Step 3.: The computer will assign the rule_counter value to the total_rule variable and close the *Graptool.rul* file.

```
/* GET THE TOTAL NUMBER OF RULES */
total_rule = rule_counter;
/* CLOSE THE Graptool.rul FILE */
fclose(rptr);
```

Step 4.: The **ReadRulebaseFromWorkingFile** will return to the **Main** function.

```
}
```



Purpose: A <u>SearchForTheWorkingRule</u> function will search for a rule in the ***rule_base*** array whose conditions have been satisfied by the ***fact_base*** array. This rule will be called a **working rule**.

Prototype:

```
char *SearchForTheWorkingRule(int *found_rule)
{
```

A found_rule is an integer variable. The found_rule will be TRUE if the **working rule** is found. Otherwise, the found_rule will be FALSE.

Pseudo-code:

Step 1.: The computer will first check the condition of *while loop* which are the values of *notdone* and *rule_ptr*. If searching for the working rule process has not been done (*notdone* is still TRUE) and the searching does not reach the end of ***rule_base*** (the *rule_ptr* value does not equal ENDARRAY character), the computer will do step two. Otherwise, the computer will go to step seven.

```
/* DO SEARCHING FOR THE WORKING RULE PROCESS */
while((notdone) && (*rule_ptr != ENDARRAY))
```

Step 2.: The computer will initialize the value of *and_result* to TRUE, and the value of *found*, *or_count*, and *or_result* to FALSE. The computer will then search for the beginning of the new rule in the ***rule_base***, which is indicated by the DEFRULE character. The address of this rule will be stored in the *rule_ptr*.

```
{
/* INITIALIZE THE VARIABLES */
and_result = TRUE;
found = or_count = or_result = FALSE;
/* SEARCH FOR THE BEGINNING Of A RULE */
rule_ptr = strchr(rule_ptr, DEFRULE);
```

Step 3.: If the computer find a new rule (*rule_ptr* value is DEFRULE character), the computer will increment the rule_counter value (rule number) by 1, assign the *rule_ptr* value to the right_rule, and clear all variable array values by assigning the ENDARRAY character to its beginning. However, if the computer does not find the new rule, the computer will go back to step one.

```
/* CHECK IF IT IS THE BEGINNING OF THE RULE */
if(*rule_ptr == DEFRULE)
 {
 /* UPDATE THE RULE NUMBER */
 rule_counter++;
 /* ASSIGN THE RULE ADDRESS TO right_rule */
 right_rule = rule_ptr;
 /* CLEAR ALL THE VALUE INSIDE variable ARRAY */
 variable[0] = ENDARRAY;
```

Step 4.: The computer does the **pattern matching** process by calling the **<u>GetAFact</u>** function to get a rule condition (stored in the ***rule_condition*** array) from the ***rule_base***. It will then do sub-step four. Otherwise (no more rule conditions or **<u>GetAFact</u>** returned ENDRF character), the computer will go to step five.

```
/* DO THE PATTERN MATCHING PROCESS */
while(*(rule_ptr = GetAFact(rule_ptr, CONDITION,
                                        rule_condition)) != (char)ENDRF)
```

Step 4.1.: The computer will separate the logical operator of that rule condition and store it in the *logical* variable. The **<u>FactIsInTheFactbase</u>** function will be called to check the existence of ***rule_condition*** in the ***fact_base*** array, the result of which, will be stored in the *match* variable. If this rule condition has a NOT operator (*logical* value equal to LOGI_NOT character), the computer will invert the value of *match* and assign a LOGI_AND character to *logical*.

```
{
/* SEPARATE THE LOGICAL OPERATOR  FROM THE RULE CONDITION */
logical = rule_condition[0];
strcpy(&rule_condition[0], &rule_condition[1]);
/*CHECK THE EXISTENCE OF THE CONDITION IN THE FACT BASE */
FactIsInTheFactbase(rule_condition, &match);
/* CHECK IF RULE CONDITION HAS A NOT OPERATOR */
if(logical == LOGI_NOT)
 {
 /* INVERTS THE match VALUE */
  match = !match;
  logical = LOGI_AND;
 }
```

Step 4.2.: If the rule condition has an OR operator (*logical* equal to LOGI_OR character), the computer will increment the value of *or_count* by 1 and then perform an **inclusive Or** between *match* and *or_result* (contains the result of previous perform **inclusive Or**). If either *or_result* or *match* is TRUE, the *or_result* will be set to TRUE and then goes to step four. Otherwise, the *or_result* will be set to FALSE and then go to step four.

```
/* CHECK IF RULE CONDITION HAS AN OR OPERATOR */
if(logical == LOGI_OR)
 {
  or_count++;
  /* DO INCLUSIVE OR BETWEEN CURRENT AND PREVIOUS OR CONDITION */
  if(or_result || match)
    or_result = TRUE;
  else
    or_result = FALSE;
 }
```

Step 4.3.: If the rule condition has an AND operator (*logical* equal to LOGI_AND character), the computer will perform an **explicit and** between *match* and *and_result* (contains the result of previous performed **explicit and**). If both *and_result* and *match* are TRUE, the *and_result* will be set to TRUE and then goes to step four. Otherwise, the *and_result* will be set to FALSE and the computer will go to step four.

```
 else
/* CHECK IF RULE CONDITION HAS THE AND OPERATOR */
 if(logical == LOGI_AND)
 {
  /* DO EXPLICIT AND BETWEEN CURRENT AND PREVIOUS AND CONDITION */
  if(and_result && match)
    and_result = TRUE;
  else
    and_result = FALSE;
 }
}
```

Step 5.: If the computer has not performed any **inclusive Or** (*or_count* value is zero), the computer will set an *or_result* to TRUE. It will do the logical operator between the logical and condition group and logical or condition group. If both groups have been satisfied by fact base (both *and_result* and *or_result* are TRUE), the *found* will be set to TRUE. Otherwise, *found* will remain FALSE.

```
/* CHECK IF ANY INCLUSIVE OR  HAS BEEN PERFORMED */
if(or_count == 0)
 or_result = TRUE;
/* CHECK IF BOTH LOGICAL GROUPS HAVE BEEN SATISFIED */
if(and_result && or_result)
 found = TRUE;
```

Step 6.: If the **working rule** is found (the *found* value is TRUE), the computer will set *notdone* to FALSE. The computer will go back to step one.

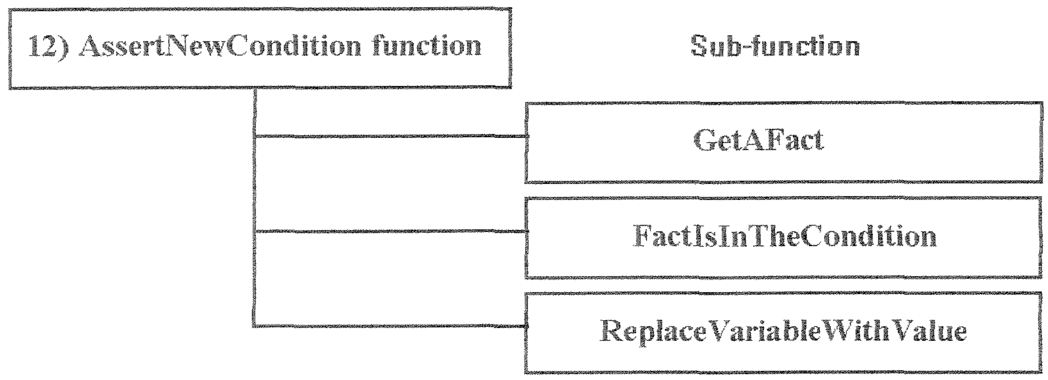
```
 /* CHECK IF THE WORKING RULE IS FOUND */
 if(found)
  notdone = FALSE;
 }
}
```

Step 7.: The computer will assign *found* value (TRUE or FALSE) to the *found_rule variable.

```
*found_rule = found;
```

Step 8.: The <u>**SearchForTheWorkingRule**</u> will return to the <u>**Main**</u> function with the value of right_rule (address of the **working rule** or the address of the ending ***rule_base*** array).

```
return(right_rule);
}
```



Purpose: An <u>AssertNewFact</u> function will add a unique **fact block** into the ***fact_base*** array.

Prototype:

```
void AssertNewFact(char *right_rule)
{
```

A right_rule is a character pointer variable which contains the address of the **working rule.**

Pseudo-code:

Step 1.: The computer will call <u>**GetAFact**</u> function to get a **new block** from the **working rule** and store it in the ***new_condition*** array. If the computer reaches the end of the **working rule** (<u>**GetAFact**</u> return an ENDRF character), it will go to step three. Otherwise, the computer will call the <u>**ReplaceVariableWithValue**</u> function to replace every **variable field** in ***new_condition*** with its value. The result of <u>**ReplaceVariableWithValue**</u> function will be stored in the ***work_str*** array. The computer will then call the <u>**FactIsInTheFactbase**</u> function to search for the **fact block** in ***fact_base*** which will match a **new block**. If the matching fact block is found, *found* will be set to TRUE.

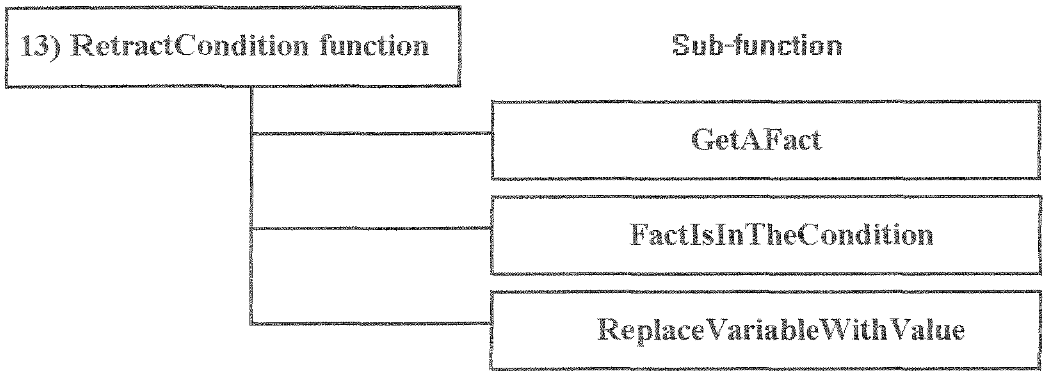
```
/* DO THE ASSERTION PROCESS */
while(*(right_rule = GetAFact(right_rule, ASSERT, new_condition)) != (char)ENDRF)
 {
 ReplaceVariableWithValue(new_condition, work_str);
 FactIsInTheFactbase(work_str, &found);
```

Step 2.: If *found* is not TRUE, the computer will copy a *work_str* to the end of ***fact_base*** array and then go back to step one. Otherwise, the computer will go back to step one without doing anything.

```
 /* CHECK IF A NEW FACT IS NOT FOUND IN THE FACT BASE */
 if(!found)
   /* ADD A NEW FACT TO THE FACT BASE */
  strcat(fact_base, work_str);
 }
```

Step 3.: The <u>**AssertNewFact**</u> will return to the <u>**Main**</u> function.

```
}
```



Purpose: A RetractOldFact function will remove a **fact block** (matches a **removing block**) from the ***fact_base*** array.

Prototype:

```
void RetractOldFact(char *right_rule)
{
```

A right_rule is a character pointer variable which contains the address of the **working rule**.

Pseudo-code:

Step 1.: The computer will call GetAFact function to get a **removing block** from the **working rule** and store it in the ***re_condition*** array. If the computer reaches the end of the **working rule** (GetAFact return an ENDRF character), the computer will go to step three. Otherwise, the computer will call the ReplaceVariableWithValue function to replace every **variable field** in ***re_condition*** with its value. The result of ReplaceVariableWithValue function will be stored in the ***work_str*** array. The computer will then call the FactIsInTheFactbase function to search for the **fact block** in ***fact_base*** which will match a **removing block**. If the matching fact block is found, its address will be stored in the *match_ptr* pointer variable, and *found* will be set to TRUE.

```
 /* DO THE RETRACTION PROCESS */
while(*(right_rule = GetAFact(right_rule, RETRACT, re_condition)) != (char)ENDRF)
 {
  ReplaceVariableWithValue(re_condition, work_str);
 match_ptr = FactIsInTheFactbase(work_str, &found);
```

Step 2.: If the matching fact block is found (the *found* value is TRUE), the computer will search for the end of the matching fact block in the ***fact_base*** array. The computer will then do step 2.1 and then go back to step one. Otherwise, the computer will go back to step one.

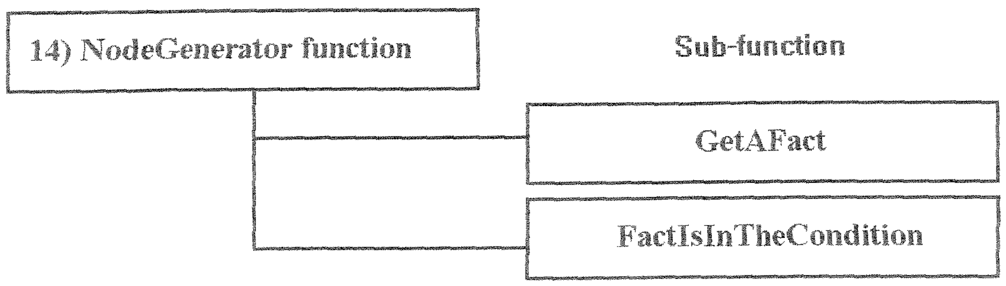
```
/* CHECK IF THE RETRACTING FACT IS FOUND */
if(found)
{
 /* SEARCH FOR THE END OF RETRACTING FACT IN THE FACT BASE */
 next_ptr = strchr((match_ptr), C_PARENTHES);
```

Step 2.1.: If the end of matching fact blocks is a closed parenthesis (end of a **block**), the computer will remove the matching fact block by coping everything after that closed parenthesis to the beginning of the matching fact block (an address is stored in the *match_ptr*). However, if the end of matching fact block is the end of ***fact_base*** array, the computer will set the beginning of matching fact block to be the end of ***fact_base***.

```
 /* REMOVE THE RETRACT FACT FROM THE FACT BASE */
 if(*next_ptr == C_PARENTHES)
  strcpy(match_ptr, ++next_ptr);
 else
  *match_ptr = ENDARRAY;
 }
}
```

Step 3.: The **RetractOldFact** will go back to the **Main** function.

```
}
```



Purpose: A <u>NodeGenerator</u> function will generate nodes which are a result of applying the logical path algorithm to test the rule-based structure. The structure of a node has already been defined in the beginning of Graptool as follows:

```
struct node
 {
  int rule_num;
  int con_num;
  char condition_set[CON_SIZE];
  struct node *ptrnext;
  struct node *worknode;
  struct node *duplicate;
 };
```

A rule_num is an integer variable which identifies the rule whose condition have been satisfied by the ***fact_base*** array.

A con_num is an integer variable which identifies the node condition set. The con_num will always be zero unless a different condition set exists with the same rule. The combination of rule_num with con_num is used for node identifiers which are called the **node number**.

A ***condition_set*** array contains a set of **fact blocks** which comes from the ***fact_base*** array.

A ptrnext is a structure pointer variable which contains the address of the next connecting node. At the last node (no connecting node), the ptrnext contains the nowhere value.

A worknode is a structure pointer variable. It contains the address of the node from which the current node was generated.

A duplicate is a structure pointer variable. It contains an existing node address which is a duplicate of the current node.

Prototype:

```
void NodeGenerator(void)
{
```

There is no prototype in this function.

Pseudo-code:

Step 1.: The computer will first assign a nowhere value to the *dupi* pointer variable, set the condition_counter variable to zero, and then generate a new node (its address is stored in the ptrnew pointer variable). If the new node is the first node (ptrfirst equals nowhere), the computer will assign the value of ptrnew to ptrfirst, ptrwork, and ptrlast and then go to step three. Otherwise, the computer will do step two.

```
/* INITIALIZE VARIABLES VALUE */
dupi = nowhere;
condition_counter = 0;
/* GENERATE THE NEW NODE */
ptrnew = (struct node *)malloc(sizeof(struct node));
/* CHECK IF THE NEW NODE IS THE FIRST NODE IN THE CHAIN */
if(ptrfirst == nowhere)
  ptrfirst = ptrwork = ptrlast = ptrnew;
```

Step 2.: The computer will assign ptrfirst value (address of the first node) to the *check* pointer variable and then do sub-step two.

```
else
  {
  check = ptrfirst;
```

Step 2.1.: The computer will check if the rule of the new node (rule number is stored in the rule_counter variable) has been used in the other nodes by checking the rule_num value of the existence node. If the rule_counter equals the rule_num value of existence node (its address is *check* value), the computer will set *match* to TRUE and go to step 2.2. Otherwise, the computer will go to step 2.3

```
  /* DO THE NEW NODE CHECKING PROCESS */
  do
   {
   /* CHECK IF RULE OF NEW NODE HAS BEEN USED IN THE OTHER NODE */
   if((rule_counter == check->rule_num)
                    && (condition_counter <= check->condition_counter))
    {
    match = TRUE;
    c_ptr = check->condition_set;
```

Step 2.2.: The computer will compare the condition set of the existing node to the **fact blocks** in the ***fact_base***. If the condition set of that existence node is the same as the **fact blocks**, the computer will assign ptrwork value to the *dupi*. Otherwise (condition set of existence node is not the same as the **fact blocks**), the computer will set the condition_counter to the con_num value of that existence node plus 1.

```
 /* CHECK THE MATCHING BETWEEN CONDITION SET AND FACT BASE */
 while((match) &&
          (*(c_ptr = GetAFact(c_ptr, DEFFACTS, work_str)) != (char)ENDARRAY))
    FactIsInTheFactbase(work_str, &match);
 /* CHECK IF CONDITION SET IS THE SAME AS THE FACT BASE */
 if((match) && (strlen(fact_base) == strlen(check->condition_set)))
   dupi = ptrwork;
 else
   condition_counter = check->con_num + 1;
 }
```

Step 2.3.: The computer will assign an address of the next node to *check*. If the *check* value does not equal nowhere (it is not the end of the node chain) and *dupi* value equals nowhere (the duplicate condition set has not been found), the computer will go back to perform step 2.1. Otherwise, the computer will assign the value of ptrnew to the ptrlast->ptrnew and ptrlast and then go to step three.

```
  /* GET AN ADDRESS OF THE NEXT CONNECTING NODE */
  check = check->ptrnext;
  }
 while((check != nowhere) && (dupi == nowhere));
 ptrlast = ptrlast->ptrnext = ptrnew;
}
```

Step 3.: The computer will initialize values of the new node components.

```
/* INITIALIZE THE NEW NODE COMPONENTS VALUE */
ptrnew->rule_num = rule_counter;
ptrnew->con_num = condition_counter;
strcpy(ptrnew->condition_set, fact_base);
ptrnew->ptrnext = nowhere;
ptrnew->worknode = ptrwork;
ptrnew->duplicate = dupi;
```

Step 4.: The <u>NodeGenerator</u> will return to the <u>Main</u> function.

```
}
```

15) DisplayTestResult function

Purpose: A **DisplayTestResult** function will create the node connection list. This list includes the node number, node condition set, and the connection between nodes.

Pseudo-code:

The computer will display the node connection list on the screen and store it in the information file. The **DisplayTestResult** will then go back to the **Main** function.

```
void DisplayTestResult(void)
{
 int i=0, letter = NONE;
 struct node *ptrname;
 if(display == nowhere)
   ptrname = ptrwork;
 else
  {
   /* GET THE CONNECTING ADDRESS */
   if(clp_flag)
    {
     ptrname = ptrlast;
     clp_flag = FALSE;
    }
   else
    ptrname = nowhere;
  }
 if(ptrname != nowhere)
  {
   printf("\n");
   fprintf(iptr, "\n");
   if(display == nowhere)
    {
     delay(DELAY_LOOP);
     printf("Working ");
     fprintf(iptr, "Working ");
    }
   printf("NODE(%d, %d)\n", ptrname->rule_num, ptrname->con_num);
   fprintf(iptr, "NODE(%d, %d)\n", ptrname->rule_num, ptrname->con_num);
   if(display == nowhere)
    {
     printf("It's condition set is...\n");
     fprintf(iptr, "It's condition set is...\n");
    }
  /*DISPLAY NODE CONDITION SET */
  while(ptrname->condition_set[i] != ENDARRAY)
   {
   letter = ptrname->condition_set[i];
   if(letter == Q_SPACE)
     letter = SPACE;
   else
   if(letter == O_PAREN)
     letter = O_PARENTHES;
   else
   if(letter == C_PAREN)
     letter = C_PARENTHES;
   printf("%c", letter);
   fprintf(iptr, "%c", letter);
   if(ptrname->condition_set[i] == C_PARENTHES)
    {
     printf("\n");
     fprintf(iptr, "\n");
    }
   +;
   }
  if(display == nowhere)
   {
   display = ptrwork;
   printf("...and connect to the following nodes: \n");
   fprintf(iptr,"...and connect to the following nodes: \n");
   }
 }
else
 {
 printf("\n TERMINATION NODE");
 fprintf(iptr, "\n TERMINATION NODE");
 }
}
```

16) FinalAnalysis function

Purpose: A **FinalAnalysis** function will do a simple rule base analysis from the node connection list.

Pseudo-code:

The computer will check how many times each rule has been used, how many nodes have been generated from each rule, and how many generated nodes were duplicated. The result of this analysis will be displayed on the screen and stored in the information file. The **FinalAnalysis** will then go back to the **Main** function.

```
void FinalAnalysis(void)
{
 int dup_count, node_count;
 rule_counter = 0;
 /* DISPLAY A PROCESS MESSAGE */
 printf("\n\n********** Rulebase Structural Analysis **********\n");
 fprintf(iptr, "\n\n********** Rulebase Structural Analysis **********\n");
 /* DO THE RULE BASE ANALYSIS PROCESSING */
 while(rule_counter < total_rule)
  {
  /* GET THE ADDRESS OF THE FIRST NODE */
  display = ptrfirst;
  dup_count = node_count = 0;
  /* DO NODE SEARCHING WHILE IT IS NOT THE END OF NODE LISTING */
  while(display != nowhere)
   {
   /* CHECK IF NODE CAME FORM THE CURRENT RULE */
   if(display->rule_num == rule_counter)
    {
    /* UPDATE THE NODE COUNTER */
    node_count++;
    /* CHECK IF THIS NODE IS A DUPLICATE NODE */
    if(display->duplicate != nowhere)
      /* UPDATE THE DUPLICATE NODE COUNTER */
      dup_count++;
    }
   /* GET AN ADDRESS OF THE NEXT NODE */
   display = display->ptrnext;
   }
  /* DISPLAY AND SAVE THE RULE BASE ANALYSIS RESULT */
  printf("\n Rule NUMBER - %d", rule_counter);
  printf("\n NUMBER OF NODES - %d", node_count);
  printf("\n NUMBER OF DUPLICATE NODES - %d \n", dup_count);
  fprintf(iptr, "\n RULE NUMBER - %d", rule_counter);
  fprintf(iptr, "\n NUMBER OF NODES - %d", node_count);
  fprintf(iptr, "\n NUMBER OF DUPLICATE NODES - %d \n", dup_count);
  delay(DELAY_LOOP);
  /* UPDATE THE RULE NUMBER */
  rule_counter++;
  }
 printf("\n **************************************************** \n");
 fprintf(iptr, "\n **************************************************** \n");
}
```

12. APPENDIX C.: GRAPTOOL SOFTWARE SOURCE CODE

```
/*******************************************************************************************
* ------------------------------------ GRAPTOOL SOURCE CODE -------------------------------- *
* ------------------------ BY MR. PIYAPATTANA TEMCHAREON ---------------------------*
*                                                                                          *
* OBJECTIVE.: Graptool is a computer tool based on the logical path graph *
*                algorithm.  Graptool will read the CLIPS rule bases as ASCII *
*                text and convert it to Graptool format.  After the conversion   *
*                is finished, Graptool will apply the logical path graph method *
*                to the Graptool format for testing of rule-based structure.     *
*                The results of this computer tool include the following:        *
*                1. Graptool.wrk which contains the CLIPS rule bases after the *
*                  Graptool has eliminated unnecessary functions or commands*
*                2. Graptool.rul which contains rules in Graptool format.        *
*                3. Graptool.fac that contains initial facts in Graptool format   *
*                4. Graptool.err (option) which contains process and error       *
*                   messages during software execution.                          *
*                5. An information file that contains results of the rule-based   *
*                   structure testing.                                           *
*                                                                                  *
* NOTE -> 1. All five files will be saved in the selected working directory.     *
*           2. Graptool will automatically stop the execution and return to       *
*              DOS operating system if it detects any error.                      *
*******************************************************************************************/
/* TURBO C++ INCLUDING FILES */
#include<dir.h>
#include<dos.h>
#include<ctype.h>
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/* DEFINE REPLACING CHARACTERS */
#define DEFRULE         14
#define ASSERT          15
#define RETRACT         16
#define CONDITION       17
#define DEFFACTS        18
#define Q_SPACE         19
#define LHSRHS          20
#define O_PAREN         21
#define C_PAREN         22
#define ENDRF           158
```

```
/* DEFINE CLIPS RULE-BASED SYMBOLS */
#define QUOTE               '"'
#define SPACE               ' '
#define ENDLHS              '='
#define COMMENT             ';'
#define LOGI_OR             '|'
#define LOGI_NOT            '~'
#define LOGI_AND            '&'
#define ENDARRAY            '\0'
#define STARTRHS            '>'
#define NEXTLINE            \n'
#define RE_POINT1           '<'
#define RE_POINT2           '-'
#define S_WILDCARD          '?'
#define M_WILDCARD          '$'
#define O_PARENTHES         '('
#define C_PARENTHES         ')'

/* DEFINE THE SIZE OF THE ARRAYS */
#define CON_SIZE            500       /* MINIMUM SIZE 500 BYTES */
#define RULE_SIZE           1024      /* MINIMUM SIZE 1024 BYTES */
#define FACT_SIZE           80        /* MINIMUM SIZE 80 BYTES */
#define PATH_SIZE           30        /* MINIMUM SIZE 30 BYTES */
#define FIELD_SIZE          60        /* MINIMUM SIZE 60 BYTES */
#define VARIABLE_SIZE       2000      /* MINIMUM SIZE 10240 BYTES */

/* DEFINE THE ACCESSORIES SYMBOLS */
#define NONE            -1
#define TRUE            1
#define FALSE           0
#define SUCCESS          0
#define DELAY_LOOP       2000

/*************************** FILE POINTERS **************************/
FILE *fptr;             /* CLIPS RULEBASE FILE          */
FILE *rptr;              /* GRAPTOOL FORMAT RULE         */
FILE *cptr;              /* GRAPTOOL FORMAT FACT_BASE */
FILE *wptr;              /* GRAPTOOL RULEBASE           */
FILE *iptr;             /* STRUCTURAL TESTING RESULT    */
FILE *eptr;              /* PROGRAM ERROR FILE           */
/**************************************************************************/

/* DEFINE ARRAYS AND POINTER */
char *rule_ptr;                      /* RULE ARRAY POINTER         */
char iname[PATH_SIZE];               /* INFORMATION FILE NAME */
char rule_base[RULE_SIZE];           /* DEFINE RULE ARRAY          */
char fact_base[CON_SIZE];            /* DEFINE FACT BASE ARRAY  */
char variable[VARIABLE_SIZE];         /* DEFINE VARIABLE ARRAY   */
```

```
/* DEFINE THE STRUCTURE OF THE NODE */
struct node
 {
  int rule_num;                          /* THE RULE NUMBER          */
  int con_num;                           /* THE CONDITION NUMBER     */
  char condition_set[CON_SIZE];          /* THE CONDITION SET        */
  struct node *ptrnext;                  /* NEXT NODE ADDRESS        */
  struct node *worknode;                 /* PARENTS NODE ADDRESS     */
  struct node *duplicate;                /* DUPLICATE NODE ADDRESS */
 };
struct node *ptrfirst, *ptrnew, *display, *ptrwork, *ptrlast, *nowhere;
int nocondition_flag, rule_counter, condition_counter, error_file_open;
int fact_base_size, rule_size, clp_flag, rule_flag, fact_flag, total_rule;

/*************************** FUNCTION PROTOTYPES ***************************/
/* 1.0 */ void ProgramIntroduction(void);
/* 2.0 */ void InitialValue(void);
/* 3.0 */ void SetWorkingDirectory(char*);
/* 3.1 */ void CurrentWorkingDirectoryIs(char*);
/* 4.0 */ void OpenTheFiles(char*);
/* 4.1 */ void ProgramInformation(char*, char*, char*);
/* 5.0 */ void ReadTheRulebaseInToWorkingFile(void);
/* 6.0 */ int ReadRuleAndFactFromWorkingFile(int);
/* 7.0 */ void ConvertRulebaseToGraptoolFormat(int);
/* 7.1 */ void ConvertConditionToGraptoolFormat(char*);
/* 7.2 */ void ConvertAssertAndRetractToGraptoolFormat(char*);
/* 7.2.1 */ void VariableSearch(int, char[], char*);
/* 7.3 */ void ConvertDeffactsToGraptoolFormat(char*);
/* 8.0 */ void PrepareInitialFacts(void);
/* 8.1 */ int ReadTheInitialFacts(void);
/* 8.2 */ void GetConditionInstruction(void);
/* 9.0 */ void ReadRulebaseFromWorkingFile(void);
/* 10.0 */ char *SearchForTheWorkingRule(int*);
/* 11.0 */ void AssertNewFact(char*);
/* 12.0 */ void RetractOldFact(char*);
/* 13.0 */ void NodeGenerator(void);
/* 14.0 */ void DisplayTestResult(void);
/* 15.0 */ void FinalAnalysis(void);
/* 16.0 */ char *WriteFormatToWorkingFile(int, char*, FILE*);
/* 17.0 */ char *FactIsInTheFactbase(char[], int*);
/* 17.1 */ int FieldUnification(char[], char[]);
/* 17.1.1 */ int LogicalOperation(char[], char[]);
/* 18.0 */ int CheckTheFieldSyntax(char*);
/* 19.0 */ void ReplaceVariableWithValue(char[], char[]);
/* 20.0 */ char *GetAField(char*, char[], int*);
/* 21.0 */ char *GetAFact(char*, int, char[]);
/* 22.0 */ void Error(char*, int, char*);
/* 23.0 */ void Message(char*, int, char*);
/***************************************************************************/
```

```
/******************************** 0.0 MAIN FUNCTION ********************************
* The Main function can be divided into four sections:                          *
* a. Initialize section is used for preparing the program. Function include     *
*    following:                                                                  *
*         1. To give a brief description of the Graptool to the end user.       *
*         2. To initialize the important variables.                              *
*         3. To set up the working directory.                                    *
*         4. To open the CLIPS rule-based file, the information file, and       *
*            the working files.                                                  *
*    Calling functions in this section are:                                      *
*         1. void ProgramIntroduction(void)                                      *
*         2. void InitialValue(void)                                             *
*         3. void SetWorkingDirectory(char*)                                     *
*         4. void OpenTheFiles(char*)                                            *
* b. Conversion section will standardize and convert the CLIPS rule bases to    *
*    Graptool format.  Function include following:                               *
*         1. To read and standardize the CLIPS rule bases.                       *
*         2. To check possible critical error.                                   *
*         3. To convert the rule bases into Graptool format.                     *
*    Calling functions in this section are:                                      *
*         1. void ReadTheRulebaseInToWorkingFile(void)                           *
*         2. int ReadRuleAndFactFromWorkingFile(int)                             *
*         3. void ConvertRulebaseToGraptoolFormat(int)                           *
* c. Preparation section is used to prepare the Graptool for the rule-based     *
*    structure testing.  Functions include following:                            *
*         1. To read the initial facts into the fact base array.                 *
*         2. To read the rules into the rule array.                              *
*    Calling functions in this section are:                                      *
*         1. void PrepareInitialFacts(void)                                      *
*         2. void ReadRulebaseFromWorkingFile(void)                              *
* d. Application section will apply the logical path graph algorithm to test    *
*    rule-based structure.  Function include the following:                      *
*         1. To search for the working rule.                                     *
*         2. To update the fact base array.                                      *
*         3. To Generate nodes and its condition set.                            *
*         4. To display the list of node connection.                             *
*         5. To do a simple rule base analysis.                                  *
*    Calling function in this section are:                                       *
*         1. char *SearchForTheWorkingRule(int*)                                 *
*         2. void AssertNewFact(char*)                                           *
*         3. void RetractOldFact(char*)                                          *
*         4. void NodeGenerator(void)                                            *
*         5. void DisplayTestResult(void)                                        *
*         6. void FinalAnalysis(void)                                            *
* The accessory functions are:                                                   *
*         1. void Error(char*, int, char*)                                       *
*         2. void Message(char*, int, char*)                                     *
*********************************************************************************/
```

```
void main(void)
{
 char path[PATH_SIZE], ans[2], *right_rule;
 int notdone = TRUE, found, found_ptr, done, assert_first, connect;
 /**** INITIALIZATION SECTION ****/
 ProgramIntroduction();
 InitialValue();
 SetWorkingDirectory(path);
 OpenTheFiles(path);
 /**** CONVERSION SECTION ****/
 ReadTheRulebaseInToWorkingFile();
 /* DISPLAY A PROCESS MESSAGE */
 Message("\n\n** Converting rule bases into Graptool format.", -1, ENDARRAY);
 /* DO GRAPTOOL FORMAT CONVERTING PROCESS */
 while(!feof(iptr))
  {
   done = ReadRuleAndFactFromWorkingFile(DEFRULE);
   ConvertRulebaseToGraptoolFormat(done);
   delay(DELAY_LOOP/2);
  }
 /* CHECK FOR THE EXISTENCE OF RULE BASE */
 if(rule_flag == FALSE)
   Error("CANNOT FIND ANY RULES IN THIS RULE BASE.", NONE, "\n");
 Message("\n\n** Converting process is finished.", NONE, "\n");
 /**** PREPARATION SECTION ****/
 PrepareInitialFacts();
 /* CHECK FOR THE EXISTENCE OF INITIAL FACT */
 if(fact_flag == FALSE)
   Message("\n!!*****!! THE INITIAL FACTS DO NOT EXIST. !!*****!!", -1, "\n");
 else
   Message("\n!!***!! EVERYTHING IS READY FOR TESTING. !!***!!",-1, "\n");
 /* DECIDE WHETHER OR NOT TO START THE TESTING PROCESS */
 printf("\n** Enter Y to start the rule-based testing process..> ");
 gets(ans);
 /* CHECK IF AN USER WANTS TO CONTINUE THE PROCESSES */
 if((strcmp(ans, "y") == SUCCESS) || (strcmp(ans, "Y") == SUCCESS))
  {
   clrscr();
   assert_first = TRUE;
   /* GET THE APPLYING ORDER OF ASSERTION AND RETRACTION */
   printf("Enter N if you want (FACT BASE - RETRACT U ASSERT), \n");
   printf("or anything else (FACT BASE U ASSERT - RETRACT)...> ");
   gets(ans);
   /* CHECK IF THE USER WANTS TO APPLY THE RETRACTION FIRST */
   if((strcmp(ans, "n") == SUCCESS) || (strcmp(ans, "N") == SUCCESS))
     assert_first = FALSE;
   Message("\n*** Starting rule base test ***\n", NONE, ENDARRAY);
   ReadRulebaseFromWorkingFile();
```

```
/* WRITE USER SELECTION TO THE INFORMATION FILE */
 if(assert_first)
   fprintf(iptr, "\n** FACT BASE SET UNION ASSERT MINUS RETRACT **\n");
 else
   fprintf(iptr, "\n** FACT BASE SET MINUS RETRACT UNION ASSERT **\n");
  /***** APPLICATION SECTION *****/
 /* INITIALIZE THE RULE-BASED TESTING VARIABLES */
 notdone = TRUE;
 display = nowhere;
 ptrwork = ptrfirst;
 rule_ptr = rule_base;
 connect = rule_counter = fact_flag = clp_flag = SUCCESS;
 NodeGenerator();
 DisplayTestResult();
 /* DO THE TESTING RULE-BASED STRUCTURE PROCESS */
 while(notdone)
  {
   /* INITIALIZE THE FACTS OF FACT BASE */
   strcpy(fact_base, ptrwork->condition_set);
   right_rule = SearchForTheWorkingRule(&found);
   /* CHECK IF RULE LHS IS SATISFIED */
   if(found)
    {
     connect = clp_flag = TRUE;
     /* CHECK THE ORDER OF ASSERTION AND RETRACTION */
     if(assert_first)
       {
        AssertNewFact(right_rule);
        RetractOldFact(right_rule);
       }
    else
       {
        RetractOldFact(right_rule);
        AssertNewFact(right_rule);
       }
     NodeGenerator();
    }
   else
    {
     /*CHECK IF WORKING NODE CONNECTS TO ANY NODE */
     if(!connect)
       {
        printf("\n!!! No CONNECTING NODE !!!\n");
        fprintf(fptr, "\n!!! No CONNECTING NODE !!!\n");
       }
     ptrwork = ptrwork->ptrnext;
```

```
        /* SEARCH FOR THE NEXT WORKING NODE */
        while((ptrwork->duplicate != nowhere) && (ptrwork != nowhere))
          ptrwork = ptrwork->ptrnext;
        if(ptrwork != nowhere)
          {
           display = nowhere;
           rule_ptr = rule_base;
           connect = rule_counter = fact_flag = SUCCESS;
          }
        else
          notdone = FALSE;
       }
      DisplayTestResult();
     }
    /* DISPLAY AN ENDING PROCESS MESSAGE */
    Message("\n\n*** Rule base test is finished ***",NONE, ENDARRAY);
    FinalAnalysis();
   }
  else
   Message("\n-- Computer processing has been terminated. --", -1, "\n");
  fcloseall();
 }
```

```
/*********************************************************************************************
* ------------------------------- 1.0 ProgramIntroduction function ------------------------------ *
* Purpose: A ProgramIntroduction function will give a general description of     *
*            the Graptool software to the end user.                              *
*                                                                                *
* Calling function: NONE                                                         *
*********************************************************************************************/
void ProgramIntroduction(void)
{
 int i;
 clrscr();
 for(i=0; i<79; i++) putch(205);
 printf("\n                              ******* GRAPTOOL.C *******\n");
 printf("PROGRAM OBJECTIVE.: Graptool is a software tool based on the ");
 printf("logical path graph algorithm.  This software will read the CLIPS ");
 printf("rule base and convert it into\nGraptool format.  After the ");
 printf("conversion is finished, the program will apply the\nlogical path ");
 printf("graph algorithm to the Graptool format for testing of rule-based\n");
 printf("structure.  The results of this tool are the following:\n");
 printf("1. GRAPTOOL.WRK contains the CLIPS rule base after Graptool has ");
 printf("eliminated the\n   unnecessary functions or commands from the ");
 printf("original rule base.\n");
 printf("2. GRAPTOOL.RUL contains rules from the rule base in Graptool ");
 printf("format.\n");
 printf("3. GRAPTOOL.FAC contains initial facts from the rule base in ");
 printf("Graptool format.\n");
 printf("4. GRAPTOOL.ERR (option) contains process and error messages ");
 printf("during software\n   execution.\n");
 printf("5. The information file contains the results of rule base testing.");
 printf("  An user can\n   select any file name except the CLIPS rule base ");
 printf("file name, and the file\n   extension cannot be '.CLP'.  If the ");
 printf("end user makes an invalid name selection,   Graptool will select a");
 printf(" unique file name which starts with 'I'.\n\n");
 printf("FINAL NOTE: All five files will be saved in the selected working ");
 printf("directory.\n");
 printf("WARNING..: COMPUTER WILL AUTOMATICALLY STOP EXECUTION IF IT FIND ");
 printf("ANY ERROR.\n");
 for(i=0; i<79; i++) putch(205);
 printf("\n\nPress any key to continue........");
 getch();
 clrscr();
}
```

```
/*******************************************************************************************
* ------------------------------------ 2.0 InitialValue function --------------------------------------- *
* Purpose: An InitialValue function initializes values of the important          *
*             variables such as the nowhere ,fact_base_size, rule_size, etc.      *
*                                                                                  *
* Calling function: NONE                                                           *
*******************************************************************************************/
void InitialValue(void)
{
 nowhere = (struct node *)NULL;
 rule_counter = condition_counter = TRUE;
 error_file_open = fact_base_size = rule_size = SUCCESS;
 rule_flag = fact_flag = clp_flag = nocondition_flag = FALSE;
}
```

```
/*********************************************************************************************
* --------------------------- 3.0 SetWorkingDirectory function -------------------------------- *
* Purpose: A SetWorkingDirectory function allows the user to select a working*
*            directory which will be used to store the working files.                    *
*                                                                                          *
* Calling functions: 1. void CurrentWorkingDirectoryIs(char*)                              *
*                    2. void Message(char*, int, char*)                                     *
*********************************************************************************************/
void SetWorkingDirectory(char *path)
{
 int disk;
 char drive[2], ans[2], *dptr;
 /* DO SET WORKING DIRECTORY PROCESSES */
 do
  {
   /* GET A CURRENT DIRECTORY */
   CurrentWorkingDirectoryIs(path);
   /* DECIDE WHETHER OR NOT TO CHANGE A WORKING DIRECTORY */
   printf("\nDo you want to change the working directory (N)? ");
   gets(ans);
   /* CHECK IF USER WANTS TO CHANGE THE WORKING DIRECTORY */
   if((strcmp(ans, "y") == SUCCESS) || (strcmp(ans, "Y") == SUCCESS))
    {
     printf("\n!! MAXIMUM PATH INCLUDING FILE NAME IS %d BYTES.",PATH_SIZE);
     /* GET A NEW WORKING DIRECTORY DRIVE */
     printf("\nEnter the drive of new working directory (A, B, C, etc.): ");
     gets(drive);
     /* GET A NEW WORKING DIRECTORY PATH */
     printf("Enter the path of new working directory (\\ for root): ");
     gets(path);
     dptr = strupr(drive);
     disk = (int)*dptr - 65;
     setdisk(disk);
     /* CHECK FOR THE EXISTENCE OF THE GIVEN DRIVE AND PATH */
     if((disk != getdisk()) || (chdir(path) != SUCCESS))
      {
       Message("\nWarning..GIVEN DRIVE OR PATH DOES NOT EXIST.", -1, "\n");
       chdir("\\");
      }
    }
  }
 /* DO PROCESSES WHILE THE USER WANTS TO CHANGE A WORKING DIRECTORY */
 while((strcmp(ans, "y") == SUCCESS) || (strcmp(ans, "Y") == SUCCESS));
}
```

```
/*******************************************************************************************
* ------------------------ 3.1 CurrentWorkingDirectoryIs function ---------------------------- *
* Purpose: A CurrentWorkingDirectoryIs function will get the current              *
*             directory information.                                              *
*                                                                                 *
* Calling function: NONE                                                          *
*******************************************************************************************/
void CurrentWorkingDirectoryIs(char *path)
{
 strcpy(path, "X:\\");
 path[0] = 'A' + getdisk();
 getcurdir(0, path + 3);
 printf("\n* THE CURRENT WORKING DIRECTORY -> %s \n", path);
}
```

```
/*******************************************************************************************
* -------------------------------- 4.0 OpenTheFiles function --------------------------------------- *
* Purpose: The OpenTheFiles function will open a CLIPS rule-based file, an       *
*            information file, an error file, and working files.                      *
*                                                                                     *
* Calling functions: 1. void ProgramInformation(char*, char*, char*)                 *
*                        2. void Error(char*, int, char*)                            *
*******************************************************************************************/
void OpenTheFiles(char *path)
{
 int length;
 char *ifile = "IXXXXXX", *wname, ans[2], cname[PATH_SIZE];
 /*GET CLIPS RULE-BASE FILE */
 printf("\nEnter the name of CLIPS rule base file -> ");
 gets(cname);
 /*GET INFORMATION FILE */
 printf("Enter the name of information file (option) -> ");
 gets(iname);
 /* DECIDE WHETHER OR NOT TO OPEN AN ERROR FILE */
 printf("Do you want to open the error file (N)? ");
 gets(ans);
 /* CONVERT FILE NAME TO CAPITAL LETTERS */
 strcpy(cname, strupr(cname));
 strcpy(iname, strupr(iname));
 /* CHECK FOR THE EXISTENCE OF CLIPS RULE-BASED FILE */
 if((fptr = fopen(cname, "r")) == NULL)
   Error("CANNOT OPEN THE CLIPS RULE BASE.", NONE, "\n");
 /* CHECK IF SELECTED INFORMATION FILE IS VALID */
 length = strlen(iname);
 if((strcmp(cname, iname) == SUCCESS) ||
     (strcmp(&iname[length-4],".CLP") == SUCCESS) ||
          (iptr = fopen(iname, "wt+")) == NULL)
  {
   Message("\nERROR!! INVALID NAME FOR THE INFORMATION FILE.",-1, "\n");
   Message("COMPUTER SELECTS AN UNIQUE NAME FOR INFORMATION FILE.",-1,"\n");
   /* GET A UNIQUE INFORMATION FILE NAME */
   wname = mktemp(ifile);
   /* RE-OPEN THE INFORMATION FILE */
   iptr = fopen(wname, "wt+");
   strcpy(iname, wname);
  }
 /* OPEN ALL THE WORKING FILES */
 wptr = fopen("GRAPTOOL.WRK", "wt+");
 rptr = fopen("GRAPTOOL.RUL", "wt+");
 cptr = fopen("GRAPTOOL.FAC", "wt+");
 /* CHECK IF USER WANTS TO OPEN AN ERROR FILE */
 if((strcmp(ans, "y") == SUCCESS) || (strcmp(ans, "Y") == SUCCESS))
  {
   error_file_open = TRUE;
   /* OPEN AN ERROR FILE */
   eptr = fopen("GRAPTOOL.ERR", "wt");
  }
 ProgramInformation(path, cname, iname);
}
```

```
/*****************************************************************************************
* ------------------------------- 4.1 ProgramInformation function ------------------------------- *
* Purpose: A ProgramInformation function provides information about Graptool *
*           software initialization such as the size of the fact array, etc.          *
*                                                                                      *
* Calling Function: void Message(char*, int, char*)                                    *
*****************************************************************************************/
void ProgramInformation(char *path, char *cname, char *iname)
{
 Message("\n*************** PROGRAM SETUP ***************",NONE, "\n");
 Message(" The fact string size is ", FACT_SIZE, " bytes.");
 Message("\n The field string size is ", FIELD_SIZE, " bytes.");
 Message("\n The rule array size is ", RULE_SIZE," bytes.");
 Message("\n The fact_base array size is ",CON_SIZE," bytes.");
 Message("\n The variable array size is ",VARIABLE_SIZE," bytes.");
 Message("\n The selected working directory is ", NONE, strupr(path));
 Message("\n The CLIPS rule base file is ", NONE, cname);
 Message("\n The information file is ",NONE, iname);
 /* CHECK IF THE ERROR FILE IS OPENED */
 if(error_file_open)
   Message("\n The error file is GRAPTOOL.ERR.", NONE, ENDARRAY);
 Message("\n**************************************************", NONE, "\n");
 printf("\nPress any key to continue..........");
 getch();
 clrscr();
}
```

```
/*****************************************************************************************
* ---------------------- 5.0 ReadTheRulebaseInToWorkingFile function ------------------- *
* Purpose: A ReadTheRulebaseInToWorkingFile function will read the CLIPS     *
*             rule-based expert system program eliminating unprintable charact-  *
*             ers, extra spaces and  comments.  It will also adjust the rule       *
*             bases, and check for critical syntax errors.  The results of this     *
*             function will be stored in an information file which will also be     *
*             used as the Graptool working file.                                    *
*                                                                                  *
* Calling functions: 1. void Error(char*, int, char*)                              *
*                         2. void Message(char*, int, char*)                       *
*****************************************************************************************/
void ReadTheRulebaseInToWorkingFile(void)
{
 div_t result;
 int ch, previous_ch=SPACE, acceptable_letter, parenthes_count, quote_count;
 Message("\n_-_ Read CLIPS rule base into the working file _-_", -1, "\0");
 /* RESET INFORMATION FILE POINTER TO THE BEGINNING OF FILE */
 rewind(iptr);
 /* SEARCH FOR THE FIRST OPEN PARENTHESIS OF THE CLIPS FILE */
 while((!feof(fptr)) && ((ch = getc(fptr)) != O_PARENTHES));
 /* CHECK IF IT IS AN OPEN PARENTHESIS */
 if(ch != O_PARENTHES)
   Error("CANNOT FIND THE BEGINNING OF THE CLIPS RULE BASE.", NONE, "\n");
 /* INITIALIZE VARIABLES VALUE */
 result.rem = parenthes_count = quote_count = SUCCESS;
 /* DO THE ADJUSTMENT AND CHECK SYNTAX ERROR PROCESS */
 while(!feof(fptr))
  {
   /* INITIALIZE acceptable_letter TO TRUE */
   acceptable_letter = TRUE;
   /* ELIMINATE AN UNPRINTABLE CHARACTER */
   if(!isprint(ch))
     acceptable_letter = FALSE;
   else
   /* CHECK IF IT IS A SPACE */
   if(ch == SPACE)
    {
     /* ELIMINATE A SPACE AFTER ANY SPACE */
     if(previous_ch == SPACE)
       acceptable_letter = FALSE;
     else
     /* ELIMINATE A SPACE AFTER THE FIRST QUOTE */
     if((previous_ch == QUOTE) && (result.rem == TRUE))
       acceptable_letter = FALSE;
     else
     /* CHECK IF IT IS OUTSIDE THE QUOTE */
     if(result.rem == SUCCESS)
      {
       /* ELIMINATE A SPACE AFTER ALL THESE CHARACTERS */
       if((previous_ch == LOGI_OR) || (previous_ch == LOGI_AND) ||
            (previous_ch == LOGI_NOT) || (previous_ch == O_PARENTHES))
          acceptable_letter = FALSE;
    else
    if(((previous_ch == S_WILDCARD) || (previous_ch == RE_POINT1) ||
          (previous_ch == ENDLHS)) && (parenthes_count == TRUE))
       acceptable_letter = FALSE;
  }
 }
else
/* CHECK IF IT IS THE BEGINNING OF THE COMMENT */
if((ch == COMMENT) && (result.rem == SUCCESS))
 {
  /* SEARCH FOR THE END OF A COMMENT */
  while((!feof(fptr)) && ((ch = getc(fptr)) != NEXTLINE));
  acceptable_letter = FALSE;
 }
else
/* CHECK IF IT IS A COMMENT AFTER THE RULE-BASED CONSTRUCT NAME */
if((ch == QUOTE) && (parenthes_count == TRUE))
 {
  /* SEARCH FOR THE END OF COMMENT */
  while((!feof(fptr)) && ((ch = getc(fptr)) != QUOTE));
  acceptable_letter = FALSE;
 }
else
/* CHECK IF IT IS A QUOTE INSIDE A BLOCK */
if((ch == QUOTE) && (parenthes_count > 1))
 {
  result = div((++quote_count), 2);
  /* ADD A SPACE TO WORKING FILE IF IT IS A FIRST QUOTE AND */
  /* AFTER A CHARACTER BESIDE AN OPEN PARENTHESIS AND A SPACE */
  if((result.rem == TRUE) &&
         (previous_ch != O_PARENTHES) && (previous_ch != SPACE))
    putc(SPACE, iptr);
 }
else
/* ADD A SPACE IN FRONT OF THESE CHARACTERS IF IT IS OUTSIDE FACT BLOCK */
if(((ch == S_WILDCARD) || (ch == ENDLHS) || (ch == RE_POINT1)) &&
      (previous_ch != SPACE) && (parenthes_count == TRUE))
  putc(SPACE, iptr);
else
/* CHECK IF IT IS AN OPEN PARENTHESIS OUTSIDE THE QUOTES */
if((ch == O_PARENTHES) && (result.rem == SUCCESS))
 {
  parenthes_count++;
  /* ADD A SPACE IN FRONT OF AN OPEN PARENTHESIS */
  if((previous_ch != SPACE) && (previous_ch != O_PARENTHES))
    putc(SPACE, iptr);
 }
else
/* SUBTRACT ONE FROM parenthes_count FOR CLOSE PARENTHESIS */
if((ch == C_PARENTHES) && (result.rem == SUCCESS))
  parenthes_count--;
  else
  /* CHECK IF IT IS AN UNIDENTIFIED CHARACTER */
  if((parenthes_count == SUCCESS) && (ch != O_PARENTHES) && (ch != SPACE))
   Error("FIND UNIDENTIFIED CHARACTERS IN THE CLIPS RULE BASE.", -1, "\n");
  /* WRITE AN ACCEPTABLE CHARACTER INTO THE INFORMATION FILE */
  if(acceptable_letter)
   {
    putc(ch, iptr);
    previous_ch = ch;
   }
  /* GET A NEW CHARACTER FROM CLIPS FILE */
  ch = getc(fptr);
 }
 /* CHECK FOR MISSING PARENTHESIS */
 if(parenthes_count != SUCCESS)
  Error("MISSING THE PARENTHESIS IN THE CLIPS RULE BASE.", NONE, "\n");
 /* CLOSING A CLIPS FILE */
 fclose(fptr);
 /* SET THE POINTER OF INFORMATION FILE TO THE BEGINNING */
 rewind(iptr);
 Message("\n\n _-_-_-_ Complete the reading _-_-_-_", NONE, "\n");
}
```

```
/******************************************************************************************
* ------------------ 6.0 ReadRuleAndFactFromWorkingFile function --------------------- *
* Purpose: A ReadRuleAndFactFromWorkingFile function will read one rule or  *
*            initial facts set at a time from the information file into a rule        *
*            array.  The computer will make adjustments and check for critical *
*            errors which it did not do in the ReadTheRulebaseInToWorkingFile  *
*            function.  The computer also compares Graptool's software config-  *
*            uration with its needs.  If the configuration does not match its       *
*            need or any errors have been detected, an appropriate error mess-  *
*            age will be displayed on the screen.  The result of this function     *
*            will be stored in the Graptool.wrk file and rule array.                  *
*                                                                                                          *
* Calling functions: 1. void Error(char*, int, char*)                                    *
*                           2. void Message(char*, int, char*)                              *
******************************************************************************************/
int ReadRuleAndFactFromWorkingFile(int type)
{
 div_t result;
 char *temptr, *currptr;
 int done, limit, fact, lhsrhs, ch, error, rule_ptr;
 int parenthes_count, quote_count, variable_count, space_count;
 int fact_size, field_size, fact_start, field_start, space_start;
 /* INITIALIZE VARIABLES VALUE */
 currptr = rule_base;
 limit = RULE_SIZE - 2;
 rule_ptr = parenthes_count = quote_count = result.rem = done = SUCCESS;
 /* SEARCH FOR THE BEGINNING OF RULE-BASED CONSTRUCT */
 while((!feof(iptr)) && ((ch = getc(iptr)) != O_PARENTHES));
 /* CHECK IF IT IS AN OPEN PARENTHESIS (BEGINNING OF CONSTRUCT) */
 if(ch == O_PARENTHES)
  {
   /* DO READING OF A RULE-BASED CONSTRUCT PROCESS */
   do
    {
     /* CHECK IF ch (BEING READ CHARACTER) IS OUTSIDE QUOTES */
     if(result.rem == SUCCESS)
      {
       /* ADD ONE TO parenthes_count FOR AN OPEN PARENTHESIS */
       if(ch == O_PARENTHES)
          parenthes_count++;
       else
       /* SUBTRACT ONE FROM parenthes_count FOR A CLOSED PARENTHESIS */
       if(ch == C_PARENTHES)
          parenthes_count--;
       /* ADD A SPACE BEHIND  LAST QUOTE */
       if((ch != SPACE) &&
         (ch != C_PARENTHES) && (rule_base[rule_ptr-1] == QUOTE))
         {
          rule_base[rule_ptr] = SPACE;
          rule_ptr++;
         }
    /* ADD A SPACE IN FRONT OF "<-" */
    if((rule_base[rule_ptr-1] == RE_POINT1) &&
      (ch == RE_POINT2) && (rule_base[rule_ptr-2] != SPACE))
      {
       rule_base[rule_ptr-1] = SPACE;
       rule_base[rule_ptr] = RE_POINT1;
       rule_ptr++;
      }
    /* DELETE SPACE BEFORE A CLOSE PARENTHESIS AND LOGICAL OPERATORS */
    if(((ch == C_PARENTHES) || (ch == LOGI_AND) || (ch == LOGI_OR)) &&
         (rule_base[rule_ptr-1] == SPACE))
        rule_ptr--;
    /* REPLACE A CLIPS ARROW WITH A LHSRHS CHARACTER */
    if((parenthes_count == TRUE) && (ch == STARTRHS) &&
         (rule_base[rule_ptr-1] == ENDLHS))
      {
       rule_ptr--;
       ch = LHSRHS;
      }
   }
  else
  /* CHECK IF ch IS BETWEEN QUOTES */
  if(result.rem == TRUE)
   {
    /* REPLACE SPACE WITH Q_SPACE CHARACTER */
    if(ch == SPACE)
         ch = Q_SPACE;
    else
    /* REPLACE OPEN PARENTHESIS WITH O_PAREN CHARACTER*/
    if(ch == O_PARENTHES)
         ch = O_PAREN;
    else
    /* REPLACE CLOSED PARENTHESIS WITH C_PAREN CHARACTER */
    if(ch == C_PARENTHES)
         ch = C_PAREN;
   }
  /* CHECK IF ch IS A QUOTE */
  if(ch == QUOTE)
   {
    /* RE-CALCULATE result.rem VALUE */
    result = div((++quote_count), 2);
    /* ELIMINATE A SPACE BEFORE THE LAST QUOTE */
    if((result.rem == SUCCESS) && (rule_base[rule_ptr-1] == Q_SPACE))
        rule_ptr--;
   }
  /* WRITE ch VALUE TO rule_base ARRAY */
  rule_base[rule_ptr] = ch;
  /* GET A NEXT CHARACTER */
  ch = getc(iptr);
 /* INCREMENT rule_ptr VALUE BY ONE */
  rule_ptr++;
 }
while((!feof(iptr)) && (parenthes_count != SUCCESS) && (rule_ptr < limit));
rule_base[rule_ptr] = ENDARRAY;
```

```
/* CHECK IF THE END OF RULE-BASED CONSTRUCT HAS NOT BEEN READ */
if(parenthes_count != SUCCESS)
 Error("THE RULE ARRAY IS TOO SMALL.", NONE, "\n");
/* SEARCH FOR THE END OF THE FIRST FIELD */
currptr = strchr(rule_base, SPACE);
/* CHECK FOR THE EXISTENCE OF THE FIRST FIELD */
if(*currptr == ENDARRAY)
 Error("CANNOT FIND ANYTHING IN THE CLIPS RULE BASE.", NONE, "\n");
*currptr = ENDARRAY;
/* CHECK IF IT IS A DEFRULE CONSTRUCT */
if(strcmp(&rule_base[1], "defrule") == SUCCESS)
 fact = FALSE;
else
/* CHECK IF IT IS A DEFFACTS CONSTRUCT */
if(strcmp(&rule_base[1], "deffacts") == SUCCESS)
 fact = TRUE;
else
 Error("CANNOT FIND DEFRULE OR DEFFACTS IN THE RULE BASE.", NONE, "\n");
/* CHECK IF COMPUTER READS THE RIGHT CONSTRUCT */
if((type == DEFFACTS) && (!fact))
 done = NONE;
else
 {
 *currptr = SPACE;
 temptr = ++currptr;
 /* SEARCH FOR THE ENDING OF rule_base ARRAY NAME */
 while((*currptr != LHSRHS) && (*currptr != O_PARENTHES) &&
   (*currptr != SPACE) && (*currptr != S_WILDCARD) && (*currptr != '\0'))
      currptr++;
 /* CHECK FOR THE EXISTENCE OF rule_base ARRAY NAME */
 if((currptr == temptr) || (*currptr == ENDARRAY))
   Error("CANNOT FIND DEFRULE NAME OR DEFFACTS NAME. ", NONE, "\n");
 ch = *currptr;
 *currptr = ENDARRAY;
 /* DISPLAY A rule_base ARRAY NAME WITH ITS TYPE */
 Message("\n\nFINISH READING.....",NONE, &rule_base[1]);
 *currptr = ch;
 /* CHECK IF IT IS A DEFRULE CONSTRUCT */
 if(!fact)
   rule_size = rule_size + rule_ptr;
 /* INITIALIZE VARIABLES VALUE */
 quote_count = space_count = variable_count = SUCCESS;
 rule_ptr =  lhsrhs = result.rem = fact_size = field_size =  SUCCESS;
 /* DO A CHECKING CONFIGURATION AND SYNTAX ERROR PROCESS */
 while(rule_base[rule_ptr] != ENDARRAY)
  {
   /* CHECK IF IT IS A SPACE */
   if(rule_base[rule_ptr] == SPACE)
     {
     /* WRITE A SPACE TO Graptool.wrk FILE */
     putc(SPACE, wptr);
     /* CALCULATE THE SPACE POSITION */
     result = div((++space_count), 2);
```

```
    /* CHECK IF A SPACE IS IN FRONT OF A FIELD */
    if(result.rem == TRUE)
      field_start = rule_ptr;
    else
    /* CALCULATE THE MAXIMUM SIZE OF FIELD */
    if(field_size < (rule_ptr - field_start + 1))
      field_size = rule_ptr - field_start + 1;
  }
else
/* CHECK  IF IT IS A QUOTE */
if(rule_base[rule_ptr] == QUOTE)
  {
   /* ADD ONE TO quote_count VALUE */
   quote_count++;
   /* ADD A QUOTE TO Graptool.wrk */
   putc(QUOTE, wptr);
  }
else
/* CHECK IF IT IS A PART OF VARIABLE FIELD */
if(rule_base[rule_ptr] == S_WILDCARD)
  {
   /* CHECK IF IT IS A PART OF RETRACT VARIABLE */
   if((parenthes_count == 1) && (rule_base[rule_ptr-2] != C_PARENTHES))
    {
     /* ADD A LINEFEED AND A SPACE TO Graptool.wrk FILE */
     putc(NEXTLINE, wptr);
     putc(SPACE, wptr);
    }
   /* UPDATE A NUMBER OF VARIABLE FIELD */
   variable_count++;
   /* ADD A QUESTION MARK TO Graptool.wrk FILE */
   putc(S_WILDCARD, wptr);
  }
else
/* CHECK IF IT IS AN OPEN PARENTHESIS */
if(rule_base[rule_ptr] == O_PARENTHES)
  {
   /* ADD ONE TO parenthes_count */
   parenthes_count++;
   fact_start = rule_ptr;
   /* CHECK IF IT IS THE BEGINNING OF FACT BLOCK */
   if((parenthes_count == 2) && (rule_base[rule_ptr-2] != RE_POINT2)
        && (rule_base[rule_ptr-2] != C_PARENTHES))
    {
     /* ADD A LINEFEED AND A SPACE TO Graptool.wrk FILE */
     putc(NEXTLINE, wptr);
     putc(SPACE, wptr);
    }
   /* ADD AN OPEN PARENTHESIS TO Graptool.wrk */
   putc(O_PARENTHES, wptr);
  }
  else
  /* CHECK IF IT IS A CLOSED PARENTHESIS */
  if(rule_base[rule_ptr] == C_PARENTHES)
    {
     parenthes_count--;
     putc(C_PARENTHES, wptr);
     /* RE-CALCULATE THE MAXIMUM FACT SIZE */
     if(fact_size < (rule_ptr - fact_start + 1))
      fact_size = rule_ptr - fact_start + 1;
     /* CHECK IF IT IS FOLLOWED BY ANOTHER CLOSED PARENTHESIS */
     if(rule_base[rule_ptr + 1] != C_PARENTHES)
      /* ADD A LINEFEED TO Graptool.wrk */
      putc(NEXTLINE, wptr);
    }
  else
  /* REPLACE A LHSRHS CHARACTER WITH => IN Graptool.wrk*/
  if(LHSRHS == rule_base[rule_ptr])
    {
     lhsrhs = TRUE;
     putc(ENDLHS, wptr);
     putc(STARTRHS, wptr);
    }
  else
  /*REPLACE A Q_SPACE WITH A SPACE IN Graptool.wrk */
  if(rule_base[rule_ptr] == Q_SPACE)
     putc(SPACE, wptr);
  else
  /*REPLACE AN O_PAREN WITH AN OPEN PARENTHESIS IN Graptool.wrk */
  if(rule_base[rule_ptr] == O_PAREN)
     putc(O_PARENTHES, wptr);
  else
  /*REPLACE A C_PAREN WITH A CLOSED PARENTHESIS IN Graptool.wrk */
  if(rule_base[rule_ptr] == C_PAREN)
     putc(C_PARENTHES, wptr);
  else
     /* WRITE A CHARACTER BEING READ INTO Graptool.wrk */
     putc(rule_base[rule_ptr], wptr);
  /* INCREMENT A rule_ptr BY ONE */
  rule_ptr++;
 }
/* ADD A LINEFEED TO GRAPTOOL.WRK */
putc(NEXTLINE, wptr);
/*CHECK IF AN ARROW IS IN DEFFACTS CONSTRUCT */
if(fact && lhsrhs)
  Error("FIND => IN THE DEFFACTS." ,NONE, "\n");
/* CHECK IF THERE IS NO ARROW IN DEFRULE CONSTRUCT */
if(!fact && !lhsrhs)
  Error("CANNOT => IN THE DEFRULE.", NONE, "\n");
result = div(quote_count, 2);
/* CHECK FOR MISSING QUOTES */
if(result.rem != SUCCESS)
  Error("MISSING QUOTES IN THE CLIPS RULE BASE.", NONE, "\n");
```

```
      /* CHECK IF A SIZE OF rule_base ARRAY IS TOO SMALL */
      if(rule_size >= limit)
        Error("RULE_SIZE, ",RULE_SIZE, ", BYTES IS TOO SMALL. \n");
      /* CHECK IF A SIZE OF fact_base ARRAY IS TOO SMALL */
      if(fact_size >= FACT_SIZE-2)
        Error("FACT_SIZE, ",FACT_SIZE, ", BYTES IS TOO SMALL. \n");
      /* CHECK IF A SIZE OF field ARRAY IS TOO SMALL */
      if(field_size >= FIELD_SIZE-2)
        Error("FIELD_SIZE, ",FIELD_SIZE, ", BYTES IS TOO SMALL. \n");
      /* CHECK IF A SIZE OF variable ARRAY IS TOO SMALL */
      if((variable_count*(field_size + fact_size)) >= VARIABLE_SIZE-2)
        Error("VARIABLE_SIZE, ",VARIABLE_SIZE,", BYTES IS TOO SMALL. \n");
      Message("\n** DOES NOT DETECT ANY ERROR IN READING PROCESS **",-1,"\0");
      done = DEFRULE;
      /* CHECK IF IT IS A DEFFACTS CONSTRUCT */
      if(fact)
        done = DEFFACTS;
    }
  }
 return(done);
}
```

```
/*******************************************************************************************
* --------------------- 7.0 ConvertRulebaseToGraptoolFormat function ------------------- *
* Purpose: A ConvertRulebaseToGraptoolFormat function will convert a defrule *
*           construct or a deffacts construct in the rule_base array into the     *
*           Graptool format. The CLIPS defrule, deffacts, assert and retract  *
*           commands will be replaced by DEFRULE, DEFFACTS, ASSERT,  *
*           RETRACT characters, respectively. Open and closed parentheses *
*           will be used as the determiner for the beginning and end of each  *
*           block, and as a name of that rule-based construct. Each rule       *
*           condition begins with a CONDITION character followed by &, |, or *
*           ~, to represent the logical operator of each condition. The end      *
*           of each defrule and deffacts construct will be marked by an ENDRF *
*           character, for separating one construct from another.               *
*                                                                                        *
* Calling functions: 1. void ConvertConditionToGraptoolFormat(char*)          *
*                    2. void ConvertAssertAndRetractToGraptoolFormat(char*)  *
*                    3. void ConvertDeffactsToGraptoolFormat(char*)              *
*                    4. char *WriteFormatToWorkingFile(int, char*, FILE*)       *
*******************************************************************************************/
void ConvertRulebaseToGraptoolFormat(int rulebase_type)
{
 char *rulebase_ptr, *temptr;
 if(rulebase_type != NONE)
  {
   /* SEARCH FOR THE BEGINNING OF RULE-BASED CONSTRUCT */
   /* NAME AND REPLACE IT WITH AN OPEN PARENTHESIS */
   temptr = strchr(rule_base, SPACE);
   *temptr = O_PARENTHES;
   /* SEARCH FOR THE END OF RULE-BASED CONSTRUCT */
   /* NAME AND REPLACE IT WITH A CLOSED PARENTHESIS */
   rulebase_ptr = strchr(temptr, SPACE);
   *rulebase_ptr = C_PARENTHES;
   rulebase_ptr++;
   /* CHECK IF IT IS A DEFRULE CONSTRUCT */
   if(rulebase_type == DEFRULE)
    {
     rule_flag = TRUE;
     /* WRITE A DEFRULE CONSTRUCT NAME TO Graptool.rul */
     WriteFormatToWorkingFile(DEFRULE, temptr, rptr);
     /* CONVERT RULE LHS INTO GRAPTOOL FORMAT */
     ConvertConditionToGraptoolFormat(rulebase_ptr);
     /* CONVERT ASSERT AND RETRACT BLOCKS INTO GRAPTOOL FORMAT */
     ConvertAssertAndRetractToGraptoolFormat(rulebase_ptr);
    }
   else
   if(rulebase_type == DEFFACTS)
    {
     /* WRITE A DEFFACTS CONSTRUCT NAME TO Graptool.fac */
     WriteFormatToWorkingFile(DEFFACTS, temptr, cptr);
     /* CONVERT FACT BLOCKS TO GRAPTOOL FORMAT */
     ConvertDeffactsToGraptoolFormat(rulebase_ptr);
    }
  }
}
```

```
/*****************************************************************************************
* ----------------- 7.1 ConvertConditionToGraptoolFormat function ------------------------*
* Purpose: A ConvertConditionToGraptoolFormat function will convert the con-  *
*            dition blocks from the LHS of the rule in the rule_base array to     *
*            Graptool format.  This conversion will be stored in Graptool.rul     *
*            file.                                                                 *
*                                                                                  *
* Calling functions: 1. void Message(char*, int, char*)                           *
*                    2. char *WriteFormatToWorkingFile(int, char*, FILE*)          *
*                    3. char *GetAField(char*, char[], int*)                       *
*****************************************************************************************/
void ConvertConditionToGraptoolFormat(char *rulebase_ptr)
{
 int condition_flag, parenthes_count, good_fact, notlast, ch;
 char ans[2], a_field[FIELD_SIZE],logical, new_logical;
 char *rulebase, *search_ptr;
 /* INITIALIZE VARIABLES */
 parenthes_count = 1;
 condition_flag = FALSE;
 Message("\nConverting condition to Graptool format...", NONE, ENDARRAY);
 /* DO THE RULE CONDITION CONVERTING PROCESS */
 while(*rulebase_ptr != LHSRHS)
  {
   /* CHECK IF A CONDITION BLOCK IS NOT INSIDE ANY LOGICAL BLOCK */
   if(parenthes_count == TRUE)
     logical = LOGI_AND;
   /* CHECK IF IT IS THE BEGINNING OF A BLOCK */
   if(*rulebase_ptr == O_PARENTHES)
    {
     /* UPDATE A PARENTHESIS COUNTER */
     parenthes_count++;
     /* RE-INITIALIZE VALUE OF good_fact */
     good_fact = FALSE;
     /* GET THE FIRST FIELD OF A BLOCK */
     rulebase = GetAField(rulebase_ptr, a_field, &notlast);
     /* CHECK IF THE END OF FIRST FIELD IS A SPACE */
     if(*rulebase == SPACE)
      {
       /* SEARCH FOR THE BEGINNING OF ANY SUB-BLOCK */
       while((*rulebase != O_PARENTHES) && (*rulebase != C_PARENTHES))
              rulebase++;
      }
     /* CHECK IF THERE IS NO SUB-BLOCK */
     if(*rulebase != O_PARENTHES)
      {
       good_fact = TRUE;
       rulebase = rulebase_ptr;
      }
      /* A BLOCK HAS SUB-BLOCK */
     else
      {
       /* UPDATE A PARENTHESIS COUNTER */
       parenthes_count++;
 /* INITIALIZE new_logical VALUE */
 new_logical = SPACE;
 /* CHECK IF A FIRST FIELD IS LOGICAL OPERATOR OR */
 if(strcmp(a_field, "or") == SUCCESS)
    new_logical = LOGI_OR;
 else
 /* CHECK IF A FIRST FIELD IS LOGICAL OPERATOR AND */
 if(strcmp(a_field, "and") == SUCCESS)
    new_logical = LOGI_AND;
 else
 /* CHECK IF A FIRST FIELD IS LOGICAL OPERATOR NOT */
 if(strcmp(a_field, "not") == SUCCESS)
    new_logical = LOGI_NOT;
 /* IF LOGICAL OPERATOR EXISTS, UPDATE THE logical */
 /* VALUE AND SET good_fact TO TRUE */
 if(new_logical != SPACE)
    {
    good_fact = TRUE;
    logical = new_logical;
    }
}
/* CHECK IF IT IS A GOOD FACT BLOCK */
if(good_fact)
 {
 condition_flag = TRUE;
 /* SET A LOGICAL OPERATOR TO THE BEGINNING OF FACT BLOCK */
 /* AND THEN WRITE THAT FACT BLOCK TO A Graptool.rul FILE */
 *(--rulebase) = logical;
 rulebase_ptr = WriteFormatToWorkingFile(CONDITION, rulebase, rptr);
 }
else
 {
 /* IF IT IS NOT A FACT BLOCK, A COMPUTER SEARCH FOR THE END */
 do
    {
    rulebase++;
    if(*rulebase == O_PARENTHES)
      parenthes_count++;
    else
    if(*rulebase == C_PARENTHES)
      parenthes_count--;
    }
 while(parenthes_count != 1);
 ch = *(++rulebase);
 *rulebase = ENDARRAY;
 /* DISPLAY A DETAIL OF A FACT BLOCK */
 Message("\n", NONE, rulebase_ptr);
 Message("\n! UNIDENTIFIED FACT BLOCK HAS BEEN ELIMINATED. !",-1,"\n");
 rulebase_ptr--;
 *rulebase_ptr = COMMENT;
```

```
      /* DO AN ELIMINATE "<-" PROCESS */
      while(*rulebase_ptr != ENDARRAY)
        {
         if((*rulebase_ptr == RE_POINT1) && (*(++rulebase_ptr) == RE_POINT2))
           *rulebase_ptr = COMMENT;
         rulebase_ptr++;
        }
      *rulebase = ch;
      rulebase_ptr = rulebase;
      /* ASK A USER TO CONFIRM PROCESS CONTINUATION */
      Message("A GRAPTOOL PROGRAM MAY NOT GIVE THE CORRECT RESULT.",-1,"\n");
      Message("ENTER Y TO CONTINUE THE PROCESS...> ",NONE, ENDARRAY);
      gets(ans);
      /* CHECK IF AN ANSWER IS NO */
      if((strcmp(ans, "Y") != SUCCESS) && (strcmp(ans, "y") != SUCCESS))
         Error("User decide to stop a program execution." , NONE, ENDARRAY);
     }
   }
  /* CHECK IF IT IS A CLOSED PARENTHESIS */
  if(*rulebase_ptr == C_PARENTHES)
    parenthes_count--;
  rulebase_ptr++;
 }
/* CHECK THE EXISTENCE OF LHS RULE */
if(!condition_flag)
 {
  nocondition_flag = TRUE;
  Message("\n!!!!!!! NO CONDITION IN THIS RULE. !!!!!!!", NONE, ENDARRAY);
  /* INSERT THE SPECIAL FACT BLOCK TO GRAPTOOL.RUL */
  putc(CONDITION, rptr);
  fprintf(rptr, "&(initial-fact)");
  putc(CONDITION, rptr);
  fprintf(rptr, "&(pt-NoCondition-TP)");
  /* INSERT THE RETRACTION OF SPECIAL FACT BLOCK TO GRAPTOOL.RUL */
  putc(RETRACT, rptr);
  fprintf(rptr,"(pt-NoCondition-TP)");
  /* CALCULATE THE TOTAL FACT_BASE SIZE */
  fact_base_size = fact_base_size + 15;
 }
}
```

```
/*****************************************************************************************
* ------------- 7.2 ConvertAssertAndRetractToGraptoolFormat function --------------- *
* Purpose: A ConvertAssertAndRetractToGraptoolFormat function will convert  *
*             the assert block and retract block of the rule base into Graptool    *
*             format.                                                               *
*                                                                                   *
* Calling functions: 1. void VariableSearch(int, char[], char*)                     *
*                    2. void Message(char*, int, char*)                             *
*                    3. char *WriteFormatToWorkingFile(int, char*, FILE*)          *
*                    4. char *GetAField(char*, char[], int*)                        *
*****************************************************************************************/
void ConvertAssertAndRetractToGraptoolFormat(char *rulebase_ptr)
{
 char a_field[FIELD_SIZE], *rulebase, *ptr;
 int notlast, assert_flag, retract_flag, flag;
 /* INITIALIZE VARIABLES VALUE */
 rulebase = rulebase_ptr;
 assert_flag = retract_flag = flag = FALSE;
 Message("\nConvert assert and retract to Graptool format...", -1, ENDARRAY);
 /* SEARCH FOR THE BEGINNING OF RHS */
 rulebase_ptr = strchr(rulebase_ptr, LHSRHS);
 /* DO THE ACTION BLOCK CONVERSION */
 while(*rulebase_ptr != ENDARRAY)
  {
   /* CHECK IF IT IS THE BEGINNING OF AN ACTION BLOCK */
   if(*rulebase_ptr == O_PARENTHES)
    {
     /* GET THE FIRST FIELD OF AN ACTION BLOCK */
     rulebase_ptr = GetAField(rulebase_ptr, a_field, &notlast);
     /* CHECK IF AN ACTION IS ASSERT OR RETRACT BLOCK */
     if((strcmp(a_field, "assert") == SUCCESS)
             || (strcmp(a_field, "retract") == SUCCESS))
      {
       /* SET THE VARIABLES FOR ASSERT BLOCK */
       if(strcmp(a_field, "assert") == SUCCESS)
         {
          rulebase_ptr++;
          flag = ASSERT;
          assert_flag = TRUE;
         }
       else
       /* SET THE VARIABLES FOR RETRACT BLOCK */
         {
          flag = RETRACT;
          retract_flag = TRUE;
         }
       ptr = rulebase_ptr;
     /* DO THE VARIABLE SEARCHING PROCESS */
     do
       {
       /* GET A FIELD FROM THE ACTION BLOCK */
       rulebase_ptr = GetAField(rulebase_ptr, a_field, &notlast);
       /*CHECK IF IT IS SINGLE-FIELD VARIABLE */
       if((a_field[0] == S_WILDCARD) && (strlen(a_field) > 1))
         VariableSearch(flag, a_field, rulebase);
       else
       /* CHECK IF IT IS A MULTIFIELD VARIABLE FROM AN ASSERT BLOCK */
       if((flag == ASSERT) && (a_field[0] == M_WILDCARD) &&
         (a_field[1] == S_WILDCARD) && (strlen(a_field) > 2))
        VariableSearch(ASSERT, a_field, rulebase);
       else
       /* CHECK IF IT IS A RETRACT BLOCK AND NO SINGLE-FIELD VARIABLE */
       if(flag == RETRACT)
        Error("RETRACT INDEX CANNOT BE MULTIFIELD VARIABLE.", -1, ENDARRAY);
       }
     while(notlast);
     /* CHECK IF ACTION BLOCK IS AN ASSERT BLOCK */
     if(flag == ASSERT)
       /* ADD ASSERT FACT STRING INTO THE GRAPTOOL.RUL FILE */
       ptr = WriteFormatToWorkingFile(ASSERT, ptr, rptr);
    }
   /* SEARCH FOR THE ENDING OF THE ACTION BLOCK */
   rulebase_ptr = strchr(rulebase_ptr, C_PARENTHES);
  }
  rulebase_ptr++;
 }
/* CHECK IF RULE HAS ANY ASSERT BLOCK */
if(!assert_flag)
  Message("\n!!!!!!!! NO ASSERT IN THIS RULE. !!!!!!!!!!", NONE, ENDARRAY);
/* CHECK IF RULE HAS ANY RETRACT BLOCK */
if(!retract_flag)
  Message("\n!!!!!!!! NO RETRACT IN THIS RULE. !!!!!!!!!!", NONE, ENDARRAY);
putc(ENDRF, rptr);
}
```

```
/*********************************************************************************************
* ------------------------------ 7.2.1 VariableSearch function ---------------------------------- *
* Purpose: A VariableSearch function will search the left hand side (LHS) of *
*            the rule in rule array for the existence of the retract index and        *
*            assert variable.                                                          *
*                                                                                      *
* Calling functions: 1. void Error(char*, int, char*)                                  *
*                    2. char *WriteFormatToWorkingFile(int, char*, FILE*)              *
*                    3. char *GetAField(char*, char[], int*)                           *
*********************************************************************************************/
void VariableSearch(int variable_type, char variable[], char *rulebase_ptr)
{
 char a_field[FIELD_SIZE], *ptr;
 int found = FALSE, parenthes_count = 1, notlast;
 /* DO THE VARIABLE SEARCHING PROCESS */
 while(*rulebase_ptr != LHSRHS)
  {
   /* CHECK IF FIRST LETTER OF variable MATCHES FIRST LETTER OF LHS FIELD */
   if(variable[0] == *rulebase_ptr)
    {
     rulebase_ptr--;
     /* GET A FIELD FROM THE LEFT HAND SIDE OF THE RULE */
     rulebase_ptr = GetAField(rulebase_ptr, a_field, &notlast);
     /* CHECK IF variable_type IS AN ASSERT CHARACTER */
     if(variable_type == ASSERT)
      {
       /* CHECK IF AN a_field CONTAINS ANY LOGICAL FIELD */
       if((ptr = strchr(a_field, LOGI_AND)) != ENDARRAY)
          /* ELIMINATE LOGICAL FIELD */
          *ptr = ENDARRAY;
          /* CHECK IF a_field MATCHES variable */
       if((strcmp(a_field, variable) == SUCCESS) && (parenthes_count > 1))
          found = TRUE;
      }
     else
     /* CHECK IF variable_type IS A RETRACT CHARACTER */
     if(variable_type == RETRACT)
      {
       /* CHECK FOR THE EXISTENCE OF variable */
       ptr = rulebase_ptr;
       if((strcmp(a_field, variable) == SUCCESS) && (*(++ptr) == RE_POINT1)
                && (*(++ptr) == RE_POINT2) && (*(++ptr) != COMMENT))
         {
          found = TRUE;
          parenthes_count++;
          /* SEARCH FOR THE END OF RETRACT FACT */
          rulebase_ptr = strchr(rulebase_ptr, O_PARENTHES);
          /* ADD RETRACT FACT TO THE Graptool.rul FILE */
          rulebase_ptr = WriteFormatToWorkingFile(RETRACT, rulebase_ptr, rptr);
         }
      }
    }
```

```
      else
      if(*rulebase_ptr == QUOTE)
        rulebase_ptr = strchr(++rulebase_ptr, QUOTE);
      if(*rulebase_ptr == O_PARENTHES)
        parenthes_count++;
      else
      if(*rulebase_ptr == C_PARENTHES)
        parenthes_count--;
      rulebase_ptr++;
    }
   if(!found)
      Error(variable, NONE, " CANNOT BE FOUND IN THIS RULE BASE. \n");
  }
```

```
/*********************************************************************************************
* -------------------- 7.3 ConvertDeffactsToGraptoolFormat function --------------------- *
* Purpose: A ConvertDeffactsToGraptoolFormat function will convert the deff-     *
*             acts construct in the rule_base array into Graptool format.  The      *
*             result of this function will be saved in the Graptool.fac file.       *
*                                                                                   *
* Calling functions: 1. void Message(char*, int, char*)                             *
*                        2. char *WriteFormatToWorkingFile(int, char*, FILE*)        *
*                        3. char *GetAFact(char*, int, char[])                       *
*                        4. int CheckTheFieldSyntax(char*)                           *
*********************************************************************************************/
void ConvertDeffactsToGraptoolFormat(char *rulebase_ptr)
{
 char a_fact[FACT_SIZE], *fact_ptr, ch;
 int good_fact, deffacts_flag = FALSE, i, parenthes_count;
 /* INITIALIZE THE VARIABLES */
 parenthes_count = 1;
 deffacts_flag = FALSE;
 Message("\nConvert deffacts to Graptool format........",NONE, ENDARRAY);
 /* DO THE DEFFACTS CONSTRUCT CONVERTING PROCESS */
 while(*rulebase_ptr != ENDARRAY)
  {
   rulebase_ptr = strchr(rulebase_ptr, O_PARENTHES);
   /* CHECK IF IT IS THE BEGINNING OF FACT */
   if(*rulebase_ptr == O_PARENTHES)
    {
     /* INITIALIZE THE VARIABLE */
     i = 0;
     parenthes_count++;
     fact_ptr = rulebase_ptr;
     a_fact[0] = O_PARENTHES;
     /* READ A FACT FROM rule_base ARRAY INTO a_fact ARRAY */
     do
      {
       +;
       rulebase_ptr++;
       /* SUBTRACT ONE IF IT IS A CLOSED PARENTHESIS */
       if(*rulebase_ptr == C_PARENTHES)
           parenthes_count--;
       else
       /* ADD ONE IF IT IS AN OPEN PARENTHESIS */
       if(*rulebase_ptr == O_PARENTHES)
           parenthes_count++;
       /* ADD FACT INTO a_fact ARRAY */
       a_fact[i] = *rulebase_ptr;
      }
     while(*rulebase_ptr != C_PARENTHES);
     a_fact[++i] = ENDARRAY;
     /* CHECK IF FACT BEING READ HAS ANY SUB-BLOCK */
     if(parenthes_count == TRUE)
      {
       /* CHECK THE FIELD SYNTAX ERROR */
       good_fact = CheckTheFieldSyntax(a_fact);
      /* CHECK IF IT IS A GOOD FACT */
      if(good_fact)
        {
         fact_flag = deffacts_flag = TRUE;
         /* ADD A GOOD FACT TO THE Graptool.fac FILE */
         WriteFormatToWorkingFile(CONDITION, a_fact, cptr);
        }
      else
        {
         Message("\n", NONE, a_fact);
         Message("\nWarning THE ABOVE FACT HAS SYNTAX ERROR.",-1, ENDARRAY);
        }
     }
    else
     {
      /* SEARCH FOR THE END OF THE BLOCK BEING READ */
      while(parenthes_count != TRUE)
        {
         rulebase_ptr++;
         if(*rulebase_ptr == O_PARENTHES)
           parenthes_count++;
         else
         if(*rulebase_ptr == C_PARENTHES)
          parenthes_count--;
        }
      ch = *(++rulebase_ptr);
      *rulebase_ptr = ENDARRAY;
      Message("\n", NONE, fact_ptr);
      Message("\nTHE UNIDENTIFIED FACT HAS BEEN ELIMINATED.", -1, ENDARRAY);
      *rulebase_ptr = ch;
     }
   }
  }
 /* CHECK IF IT HAS NO INITIAL FACT */
 if(!deffacts_flag)
  Message("\n!!!!!! NO GOOD INITIAL FACT IN THIS DEFFACTS. !!!!!!", -1, "\n");
 else
   putc(ENDRF, cptr);
}
```

```
/*******************************************************************************************
* ------------------------------ 8.0 PrepareInitialFacts function -------------------------------- *
* Purpose: A PrepareInitialFacts function will prepare the initial facts for     *
*          the rule-based structure testing process.  If initial facts alre-     *
*          ady exist in the Graptool.fac file, the computer will read that       *
*          initial facts into the fact_base array.  The end user may also        *
*          add extra facts by keyboard.  However, if initial facts do not        *
*          exist, the end user has two choices:  to enter the name of the        *
*          file which has the initial facts, or to enter all the facts by        *
*          hand.                                                                 *
*                                                                                *
* Calling functions: 1. int ReadTheInitialFacts(void)                            *
*                    2. void GetConditionInstruction(void)                       *
*                    3. void ReadTheRulebaseInToWorkingFile(void)                *
*                    4. int ReadRuleAndFactFromWorkingFile(int)                  *
*                    5. void ConvertRulebaseToGraptoolFormat(int)                *
*                    6. void Message(char*, int, char*)                          *
*                    7. int CheckTheFieldSyntax(char*)                           *
*                    8. char *FactIsInTheFactbase(char[], int*)                  *
*******************************************************************************************/
void PrepareInitialFacts(void)
{
 div_t result;
 int i, notdone, limit, fact_len, fact_ptr, ch;
 int space_count, field_start, good_fact, parenthes_count;
 int con_ptr, done, quote_count, acceptable_letter, field_size;
 char fact[FACT_SIZE], deffacts_file[PATH_SIZE], ans[2];
 /* CHECK IF INITIAL FACTS ALREADY EXIST */
 if(!fact_flag)
  {
   printf("\n!!!!!!!!!! THE INITIAL FACT DOES NOT EXIST !!!!!!!!!");
   printf("\n! If you want to enter facts by hand, press enter  !");
   printf("\n! otherwise enter the deffacts file name............!\n");
   printf("\nPATH AND FILE NAME CANNOT BE LONGER THAN %d BYTES." , PATH_SIZE);
   printf("\nEnter the deffacts file name -> ");
   /* GET THE END USER SELECTION */
   gets(deffacts_file);
   /* CHECK IF  THE SELECTED FILE CAN BE OPENED */
   if((fptr = fopen(deffacts_file, "r")) != NULL)
    {
     Message("\n@@@@ THE FACT FILE IS ", NONE, deffacts_file);
     iptr = fopen(iname, "wt+");
     ReadTheRulebaseInToWorkingFile();
     /* DO GRAPTOOL FORMAT CONVERSION PROCESS */
     while(!feof(iptr))
      {
       done = ReadRuleAndFactFromWorkingFile(DEFFACTS);
       /* CHECK IF IT IS A DEFFACTS CONSTRUCT */
       if(done == DEFFACTS)
          ConvertRulebaseToGraptoolFormat(done);
      }
    }
  }
```

```
/* CHECK IF ANY OF THE RULES HAVE NO CONDITION */
if(nocondition_flag)
  /* COPY A SPECIAL FACT BLOCK TO FACT BASE */
  strcpy(fact_base, "(pt-NoCondition-TP)");
else
  /* CLEAR ALL THE VALUES IN THE FACT BASE */
  fact_base[0] = ENDARRAY;
/* CHECK IF INITIAL FACTS ALREADY EXIST */
if(fact_flag)
 {
  /* READ INITIAL FACTS INTO THE FACT BASE */
  con_ptr = ReadTheInitialFacts();
  printf("\nDo you want to add the extra facts to fact base (N)? ");
  gets(ans);
 }
/* THE INITIAL FACT DOES NOT EXIST */
else
 {
  strcpy(ans, "Y");
  /* GET THE ENDING ADDRESS OF fact_base ARRAY */
  con_ptr = strlen(fact_base);
  Message("\n\nYOU WILL ENTER ALL THE INITIAL FACTS BY HAND." ,NONE, "\n");
 }
/* DO THE ENTERING INITIAL FACTS BY HAND PROCESS */
while((strcmp(ans, "Y") == SUCCESS) || (strcmp(ans, "y") == SUCCESS))
 {
  /* INITIALIZE THE PROCESSING VARIABLES */
  i = con_ptr;
  notdone = TRUE;
  limit = CON_SIZE - 2;
  fact_base[i] = ENDARRAY;
  GetConditionInstruction();
  /* DO THE GETTING AND CONVERTING INITIAL FACT PROCESS */
  while(notdone)
   {
    /* INITIALIZE THE PROCESSING VARIABLES */
    fact_len = FACT_SIZE - 2;
    parenthes_count = fact_ptr = quote_count = result.rem = SUCCESS;
    printf("\n!! FACT STRING CANNOT BE LONGER THAN %d BYTES. !!",FACT_SIZE);
    printf("\n!! EACH FIELD CANNOT BE LONGER THAN %d BYTES.  !!",FIELD_SIZE);
    printf("\n** You have %d bytes left in the fact_base array. **",limit-i);
    printf("\nEnter the fact string ........");
    printf("\n12345678901234567890123456789012345678901234567890123456789O");
    printf("1234567890123456789\n");
    /* GET AN INITIAL FACT FROM THE KEYBOARD */
    while((fact_ptr < fact_len) && ((ch = getch()) != '\r'))
     {
      acceptable_letter = TRUE;
      /* ENTER THE UNPRINTABLE CHARACTER */
      if(!isprint(ch))
         acceptable_letter = FALSE;
      else
   /* CHECK IF ch IS A SPACE */
   if(ch == SPACE)
   {
    /* ELIMINATE SPACES AFTER SPACE */
    if(fact[fact_ptr-1] == SPACE)
      acceptable_letter = FALSE;
    else
    /* ELIMINATE SPACES AFTER FIRST QUOTE */
    if((fact[fact_ptr-1] == QUOTE) && (result.rem == TRUE))
      acceptable_letter = FALSE;
    else
    /* ELIMINATE SPACES AFTER OPEN PARENTHESIS OUTSIDE THE QUOTE */
    if((fact[fact_ptr-1] == O_PARENTHES) && (result.rem == SUCCESS))
      acceptable_letter = FALSE;
   }
else
/* CHECK IF ch IS A QUOTE */
if(ch == QUOTE)
   {
    /* CALCULATE THE QUOTE AND ch POSITION */
    result = div((++quote_count), 2);
    /* ELIMINATE SPACE AFTER THE SECOND QUOTE */
    if((result.rem == SUCCESS) && (fact[fact_ptr-1] == SPACE))
      fact_ptr--;
    else
    /* ADD A SPACE IN FRONT OF THE FIRST QUOTE IF IT DOSE */
    /* NOT FOLLOW AN OPEN PARENTHESIS AND A SPACE */
    if((result.rem == TRUE) &&
      (fact[fact_ptr-1] != SPACE) && (fact[fact_ptr-1] != O_PARENTHES))
     {
      fact[fact_ptr] = SPACE;
      printf("%c", SPACE);
      fact_ptr++;
     }
   }
else
/* ELIMINATE A SPACE AFTER THE CLOSED PARENTHESIS OUTSIDE THE QUOTE */
if((ch == C_PARENTHES) &&
        (result.rem == SUCCESS) && (fact[fact_ptr-1] == SPACE))
    fact_ptr--;
else
/* ADD A SPACE AFTER THE SECOND QUOTE IF IT IS */
/* NOT FOLLOWED BY A CLOSED PARENTHESIS AND A SPACE */
if((ch != SPACE) && (ch != C_PARENTHES) &&
        (result.rem == SUCCESS) && (fact[fact_ptr-1] == QUOTE))
   {
    fact[fact_ptr] = SPACE;
    printf("%c", SPACE);
    fact_ptr++;
   }
/* UPDATE THE PARENTHESIS COUNTER */
if(((ch == O_PARENTHES) ||
      (ch == C_PARENTHES)) && (result.rem == SUCCESS))
    parenthes_count++;
```

```
  /* CHECK IF ch IS AN ACCEPTABLE CHARACTER */
  if(acceptable_letter)
    {
     /* ADD ch TO THE fact ARRAY AND DISPLAY IT ON THE SCREEN */
     fact[fact_ptr] = (char)ch;
     printf("%c", ch);
     fact_ptr++;
    }
  }
fact[fact_ptr] = ENDARRAY;
/* CHECK IF THE FIRST CHARACTER IN fact IS AN OPEN PARENTHESIS */
if(fact[0] != O_PARENTHES)
    notdone = FALSE;
else
 {
  Message("\nProcessing initial fact ........\n" ,NONE, ENDARRAY);
  fact_len = strlen(fact) - 1;
  /* CHECK IF THE INITIAL FACT IS MISSING A QUOTE */
  if(result.rem != SUCCESS)
     Message("\nERROR! MISSING QUOTE IN THE INITIAL FACT." ,NONE, "\n");
  else
  /* CHECK IF THE INITIAL FACT IS MISSING CLOSED PARENTHESIS */
  if(fact[fact_len] != C_PARENTHES)
    Message("\nERROR! MISSING CLOSED PARENTHESIS AT THE END." ,-1,"\n");
  else
  /* CHECK IF INITIAL FACT HAS A SUB-FACT */
  if(parenthes_count > 2)
     Message("\nERROR !!!! UNIDENTIFIED INITIAL FACT !!!!",NONE, "\n");
  else
  /* CHECK THE INITIAL FACT SIZE */
  if((i + fact_len) >= limit)
     Message("\nERROR! FACT IS TOO LONG FOR THE FACT BASE.",-1,"\n");
  else
    {
     quote_count = result.rem = SUCCESS;
     /* DO THE REPLACING CHARACTER PROCESS */
     for(fact_ptr = 0; fact_ptr <= fact_len; fact_ptr++)
      {
       /* CHECK IF IT IS A QUOTE */
       if(fact[fact_ptr] == QUOTE)
        /* UPDATE THE CHARACTER POSITION */
        result = div((++quote_count), 2);
       /* CHECK IF A CHARACTER IS BETWEEN QUOTES */
       if(result.rem == TRUE)
        {
        /* REPLACE SPACES WITH Q_SPACE CHARACTER */
        if(fact[fact_ptr] == SPACE)
          fact[fact_ptr] = Q_SPACE;
        else
        /* REPLACE OPEN PARENTHESIS WITH O_PAREN CHARACTER */
        if(fact[fact_ptr] == O_PARENTHES)
              fact[fact_ptr] = O_PAREN;
        else
        /* REPLACE CLOSED PARENTHESIS WITH C_PAREN CHARACTER */
        if(fact[fact_ptr] == C_PARENTHES)
            fact[fact_ptr] = C_PAREN;
       }
      }
      space_count = field_start = field_size = result.rem = SUCCESS;
      /* CALCULATE THE MAXIMUM FIELD SIZE OF THE GIVEN FACT */
      for(fact_ptr = 0; fact_ptr <= fact_len; fact_ptr++)
       {
        if(ch == SPACE)
         {
          result = div((++space_count), 2);
          if(result.rem == TRUE)
            field_start = fact_ptr;
          else
          if(field_size < (fact_ptr - field_start + 1))
            field_size = fact_ptr - field_start + 1;
         }
       }
      if(field_size == SUCCESS)
        field_size = fact_len + 1;
      /* CHECK IF IT EXCEEDS AN INITIALIZE FIELD SIZE */
      if(field_size >= FIELD_SIZE - 2)
        Message("\nERROR! FIELD IS TOO BIG FOR THE FIELD ARRAY.",-1,"\n");
      else
       {
        /* CHECK FOR THE FIELD SYNTAX ERROR */
        good_fact = CheckTheFieldSyntax(fact);
        /* CHECK IF IT IS AN ERROR FIELD */
        if(!good_fact)
          Message("\nERROR! GIVEN FACT HAS SYNTAX ERROR.", NONE, "\n");
        else
         {
          /* CHECK FOR THE EXISTENCE OF INITIAL FACT IN THE FACT BASE */
          FactIsInTheFactbase(fact, &good_fact);
          /* CHECK IF FIELD EXISTS IN THE FACT BASE */
          if(good_fact)
            Message("\n", NONE, "ALREADY EXISTS. <!!>\n");
          else
           {
            /* ADD THE INITIAL FACT INTO THE fact_base ARRAY */
            Message("\n", NONE, "HAS BEEN ACCEPTED. <**>\n");
            strcat(fact_base, fact);
            i = strlen(fact_base);
           }
         }
       }
     }
  }
}
```

```
   printf("\n\n@@@ ALL INPUT FACTS WILL BE DELETED IF YOU SELECT 'Y'. @@@\n");
   printf("Do you want to re-enter the facts in fact base (N)? ");
   gets(ans);
  }
 if((strlen(fact_base) > 0)
         && (strcmp(fact_base,"(pt-NoCondition-TP)") != SUCCESS))
   fact_flag = TRUE;
}
```

```
/*******************************************************************************************
* -------------------------------- 8.1 ReadTheInitialFacts function -------------------------------- *
* Purpose: A ReadTheInitialFacts function will read the initial facts from        *
*             the Graptool.fac file into the fact_base array.                      *
*                                                                                  *
* Calling functions: 1. void Message(char*, int, char*)                            *
*                    2. char *FactIsInTheFactbase(char[], int*)                    *
*******************************************************************************************/
int ReadTheInitialFacts(void)
{
 int ch, found, con_ptr, i;
 char deffacts_name[FACT_SIZE];
 Message("\n<<<< Start reading facts from Graptool.fac >>>>\n",-1, ENDARRAY);
 /* SET Graptool.fac FILE POINTER TO THE BEGINNING */
 rewind(cptr);
 /* GET A CURRENT ENDING OF fact_base ARRAY */
 con_ptr = strlen(fact_base);
 /* DO THE INITIAL FACT READING PROCESS */
 while(!feof(cptr))
  {
   ch = getc(cptr);
   /* CHECK IF IT IS THE BEGINNING OF INITIAL FACT SET */
   if(ch == DEFFACTS)
    {
     i = 0;
     ch = getc(cptr);
     /* GET THE NAME OF THE INITIAL FACT SET */
     while((ch = getc(cptr)) != C_PARENTHES)
      {
       deffacts_name[i] = ch;
       +;
      }
     deffacts_name[i] = ENDARRAY;
     /* DO THE READ INITIAL FACT INTO fact_base ARRAY */
     while(((ch = getc(cptr)) != ENDRF) && (!feof(cptr)))
      {
       if(ch != CONDITION)
          {
           /* READ THE INITIAL FACTS INTO THE FACT_BASE ARRAY */
           fact_base[con_ptr] = ch;
           con_ptr++;
          }
      }
     /* CHECK IF IT IS THE END OF INITIAL FACT SET */
     if(ch == ENDRF)
       /* DISPLAY THE NAME OF INITIAL FACT SET */
       Message("\nFINISH READING DEFFACTS...", NONE, deffacts_name);
    }
  }
 fact_base[con_ptr] = ENDARRAY;
 /* CLOSE THE Graptool.fac FILE */
 fclose(cptr);
 FactIsInTheFactbase("(initial-fact)", &found);
 /* CHECK IF (initial-fact) IS NOT IN fact_base ARRAY */
 if(!found)
  /* ADD (initial-fact) TO THE END OF fact_base ARRAY */
  strcat(fact_base,"(initial-fact)");
 Message("\n\n<<<< End reading facts from the Graptool.fac >>>>\n",-1,"\0");
 /* RE-CALCULATE THE END ADDRESS OF fact_base ARRAY */
 con_ptr = strlen(fact_base);
 return(con_ptr);
}
```

```
/*********************************************************************************************
* ---------------------------- 8.2 GetConditionInstruction function ----------------------------- *
* Purpose: A GetConditionInstruction function will provide instructions on      *
*             how to enter initial facts by keyboard to the Graptool software.      *
*                                                                                  *
* Calling function: NONE                                                           *
*********************************************************************************************/
void GetConditionInstruction(void)
{
 printf("\n***************** IMPORTANT INSTRUCTIONS ********************");
 printf("\n The size of the fact_base array is %d bytes. ",CON_SIZE);
 printf("\n EVERY INITIAL FACT MUST START AND END WITH A PARENTHESES.");
 printf("\n Otherwise the computer will not accept the given fact.");
 printf("\n PRESS RETURN (ENTER) AFTER FINISHING INPUT A INITIAL FACT.");
 printf("\n Otherwise the computer will not start the fact process.");
 printf("\n The computer will accept ONLY ONE INITIAL FACT AT A TIME.");
 printf("\n The syntax of the fact is the SAME as the fact of CLIPS.");
 printf("\n**************************************************************\n");
}
```

```
/****************************************************************************************
* ------------------------ 9.0 ReadRulebaseFromWorkingFile function --------------------- *
* Purpose: A ReadRulebaseFromWorkingFile function will read the rule bases  *
*              from the Graptool.rul file into the rule_base array.                    *
*                                                                                       *
* Calling function: NONE                                                                *
****************************************************************************************/
void ReadRulebaseFromWorkingFile(void)
{
 int ch, i=0;
 /* SET Graptool.rul FILE POINTER TO THE BEGINNING */
 rewind(rptr);
 /* SET INFORMATION FILE POINTER TO THE BEGINNING */
 rewind(iptr);
 printf("\nRule number 0 is the initial-state. \n");
 fprintf(iptr, "Rule number 0 is the initial-state. \n");
 /* DO THE RULE BASE READING PROCESS */
 while(!feof(rptr))
  {
   ch = getc(rptr);
   /*CHECK IF IT IS THE BEGINNING OF RULE */
   if(ch == DEFRULE)
    {
     ch = getc(rptr);
     /* DISPLAY THE NUMBER OF RULES */
     printf("Rule number %d is the ", rule_counter);
     fprintf(iptr ,"Rule number %d is the ", rule_counter);
     /* GET AND DISPLAY RULE BASE NAME */
     while((ch = getc(rptr)) != C_PARENTHES)
      {
       printf("%c", ch);
       fprintf(iptr, "%c", ch);
      }
     rule_base[i++] = DEFRULE;
     printf(".\n");
     fprintf(iptr, ".\n");
     /* DO THE READING RULE PROCESS */
     while((ch = getc(rptr)) != ENDRF)
       /* WRITE A RULE INTO THE rule_base ARRAY */
       rule_base[i++] = ch;
     rule_base[i++] = ENDRF;
     rule_base[i] = ENDARRAY;
     /* INCREMENT THE RULE NUMBER BY ONE */
     rule_counter++;
    }
  }
 /* GET THE TOTAL NUMBER OF RULES */
 total_rule = rule_counter;
 /* CLOSE THE Graptool.rul FILE */
 fclose(rptr);
}
```

```
/*****************************************************************************************
* ---------------------- 10.0 SearchForTheWorkingRule function -----------------------------*
* Purpose: A SearchForTheWorkingRule function will search for a rule in the      *
*             rule_base array whose conditions have been satisfied by the fact  *
*             base.  This rule will be called a working rule.                    *
*                                                                                *
* Calling functions: 1. char *GetAFact(char*, int, char[])                       *
*                         2. char *FactIsInTheFactbase(char[], int*)             *
*****************************************************************************************/
char *SearchForTheWorkingRule(int *found_rule)
{
 char rule_condition[FACT_SIZE], *right_rule, logical;
 int found, notdone = TRUE, match, and_result, or_result, or_count;
 /* DO SEARCHING FOR THE WORKING RULE PROCESS */
 while((notdone) && (*rule_ptr != ENDARRAY))
  {
   /* INITIALIZE THE VARIABLES */
   and_result = TRUE;
   found = or_count = or_result = FALSE;
   /* SEARCH FOR THE BEGINNING Of A RULE */
   rule_ptr = strchr(rule_ptr, DEFRULE);
   /* CHECK IF IT IS THE BEGINNING OF THE RULE */
   if(*rule_ptr == DEFRULE)
    {
     /* UPDATE THE RULE NUMBER */
     rule_counter++;
     /* ASSIGN THE RULE ADDRESS TO right_rule */
     right_rule = rule_ptr;
     /* CLEAR ALL THE VALUE INSIDE variable ARRAY */
     variable[0] = ENDARRAY;
     /* DO THE PATTERN MATCHING PROCESS */
     while(*(rule_ptr = GetAFact(rule_ptr, CONDITION,
            rule_condition)) != (char)ENDRF)
      {
       /* SEPARATE THE LOGICAL OPERATOR  FROM THE RULE CONDITION */
       logical = rule_condition[0];
       strcpy(&rule_condition[0], &rule_condition[1]);
       /*CHECK THE EXISTENCE OF THE CONDITION IN THE FACT BASE */
       FactIsInTheFactbase(rule_condition, &match);
       /* CHECK IF RULE CONDITION HAS A NOT OPERATOR */
       if(logical == LOGI_NOT)
         {
          /* INVERTS THE match VALUE */
          match = !match;
          logical = LOGI_AND;
         }
```

```
        /* CHECK IF RULE CONDITION HAS AN OR OPERATOR */
        if(logical == LOGI_OR)
          {
           or_count++;
           /* DO INCLUSIVE OR BETWEEN CURRENT AND PREVIOUS OR CONDITION */
           if(or_result || match)
             or_result = TRUE;
           else
             or_result = FALSE;
          }
        else
        /* CHECK IF RULE CONDITION HAS THE AND OPERATOR */
        if(logical == LOGI_AND)
          {
           /* DO EXPLICIT AND BETWEEN CURRENT AND PREVIOUS AND CONDITION */
           if(and_result && match)
             and_result = TRUE;
           else
             and_result = FALSE;
          }
      }
      /* CHECK IF ANY INCLUSIVE OR  HAS BEEN PERFORMED */
      if(or_count == 0)
        or_result = TRUE;
      /* CHECK IF BOTH LOGICAL GROUPS HAVE BEEN SATISFIED */
      if(and_result && or_result)
        found = TRUE;
      /* CHECK IF THE WORKING RULE IS FOUND */
      if(found)
        notdone = FALSE;
    }
  }
 *found_rule = found;
 return(right_rule);
}
```

```
/*****************************************************************************************
* --------------------------------- 11.0 AssertNewFact function --------------------------------- *
* Purpose: An AssertNewFact function will add a unique new fact into the     *
*             fact base.                                                      *
*                                                                             *
* Calling functions: 1. char *GetAFact(char*, int, char[])                   *
*                         2. char *FactIsInTheFactbase(char[], int*)          *
*                         3. void ReplaceVariableWithValue(char[], char[])    *
*****************************************************************************************/
void AssertNewFact(char *right_rule)
{
 int found;
 char new_condition[FACT_SIZE], work_str[VARIABLE_SIZE + FACT_SIZE];
 /* DO THE ASSERTION PROCESS */
 while(*(right_rule = GetAFact(right_rule, ASSERT, new_condition)) != (char)ENDRF)
  {
   ReplaceVariableWithValue(new_condition, work_str);
   FactIsInTheFactbase(work_str, &found);
   /* CHECK IF A NEW FACT IS NOT FOUND IN THE FACT BASE */
   if(!found)
    /* ADD A NEW FACT TO THE FACT BASE */
    strcat(fact_base, work_str);
  }
}
```

```
/*******************************************************************************************
* ------------------------------------- 12.0 RetractOldFact function ----------------------------- *
* Purpose: A RetractOldFact function will remove a fact from the fact base.   *
*                                                                              *
* Calling functions: 1. char *GetAFact(char*, int, char[])                     *
*                    2. char *FactIsInTheFactbase(char[], int*)                 *
*                    3. void ReplaceVariableWithValue(char[], char[])           *
*******************************************************************************************/
void RetractOldFact(char *right_rule)
{
 int found;
 char re_condition[FACT_SIZE], *next_ptr;
 char work_str[VARIABLE_SIZE + FACT_SIZE], *match_ptr;
 /* DO THE RETRACTION PROCESS */
 while(*(right_rule = GetAFact(right_rule, RETRACT, re_condition)) != (char)ENDRF)
  {
   ReplaceVariableWithValue(re_condition, work_str);
   match_ptr = FactIsInTheFactbase(work_str, &found);
   /* CHECK IF THE RETRACTING FACT IS FOUND */
   if(found)
    {
     /* SEARCH FOR THE END OF RETRACTING FACT IN THE FACT BASE */
     next_ptr = strchr((match_ptr), C_PARENTHES);
     /* REMOVE THE RETRACT FACT FROM THE FACT BASE */
     if(*next_ptr == C_PARENTHES)
      strcpy(match_ptr, ++next_ptr);
     else
      *match_ptr = ENDARRAY;
    }
  }
}
```

```
/*****************************************************************************************
* ------------------------------- 13.0 NodeGenerator function ---------------------------------- *
* Purpose: A NodeGenerator function will generate nodes which are a result    *
*            of applying the logical path algorithm to test the rule-based           *
*            structure.  The structure of a node has already been defined in      *
*            the beginning of Graptool as follows:                                          *
*            1. A rule_num is an integer variable which identifies the rule          *
*              whose condition have been satisfied by the fact_base array.      *
*            2. A con_num is an integer variable which identifies the node         *
*              condition set.  The con_num will always be zero unless a diff-    *
*              erent condition set exists with the same rule.  The combinat-     *
*              ion of rule_num with con_num is used for node identifiers          *
*              which are called the node number.                                           *
*            3. A condition_set array contains a set of fact blocks which          *
*              comes from the fact_base array.                                               *
*            4. A ptrnext is a structure pointer variable which contains the       *
*              address of the next connecting node.  At the last node (no          *
*              connecting node), the ptrnext contains the nowhere value.         *
*            5. A worknode is a structure pointer variable.  It contains the        *
*             address of the node from which the current node was generated.  *
*            6. A duplicate is a structure pointer variable.  It contains an         *
*              existing node address which is a duplicate of the current node   *
*                                                                                                             *
* Calling functions: 1. char *GetAFact(char*, int, char[])                                *
*                        2. char *FactIsInTheFactbase(char[], int*)                          *
*****************************************************************************************/
void NodeGenerator(void)
{
 int match;
 struct node *dupi, *check;
 char work_str[FACT_SIZE + VARIABLE_SIZE], *c_ptr;
 /* INITIALIZE VARIABLES VALUE */
 dupi = nowhere;
 condition_counter = 0;
 /* GENERATE THE NEW NODE */
 ptrnew = (struct node *)malloc(sizeof(struct node));
 /* CHECK IF THE NEW NODE IS THE FIRST NODE IN THE CHAIN */
 if(ptrfirst == nowhere)
   ptrfirst = ptrwork = ptrlast = ptrnew;
 else
  {
   check = ptrfirst;
   /* DO THE NEW NODE CHECKING PROCESS */
   do
    {
     /* CHECK IF RULE OF NEW NODE HAS BEEN USED IN THE OTHER NODE */
     if((rule_counter == check->rule_num)
                                  && (condition_counter <= check->condition_counter))
      {
       match = TRUE;
       c_ptr = check->condition_set;
      /* CHECK THE MATCHING BETWEEN CONDITION SET AND FACT BASE */
      while((match) &&
          (*(c_ptr = GetAFact(c_ptr, DEFFACTS, work_str)) != (char)ENDARRAY))
              FactIsInTheFactbase(work_str, &match);
      /* CHECK IF CONDITION SET IS THE SAME AS THE FACT BASE */
      if((match) && (strlen(fact_base) == strlen(check->condition_set)))
          dupi = ptrwork;
      else
          condition_counter = check->con_num + 1;
     }
    /* GET AN ADDRESS OF THE NEXT CONNECTING NODE */
    check = check->ptrnext;
   }
  while((check != nowhere) && (dupi == nowhere));
  ptrlast = ptrlast->ptrnext = ptrnew;
 }
 /* INITIALIZE THE NEW NODE COMPONENTS VALUE */
 ptrnew->rule_num = rule_counter;
 ptrnew->con_num = condition_counter;
 strcpy(ptrnew->condition_set, fact_base);
 ptrnew->ptrnext = nowhere;
 ptrnew->worknode = ptrwork;
 ptrnew->duplicate = dupi;
}
```

```
/*****************************************************************************************
* ---------------------------- 14.0 DisplayTestResult function -------------------------------- *
* Purpose: A DisplayTestResult function will create node connection list.      *
*              This list includes the node number, node condition set, and the    *
*              connection between nodes.                                            *
*                                                                                   *
* Calling function: NONE                                                            *
*****************************************************************************************/
void DisplayTestResult(void)
{
 int i=0, letter = NONE;
 struct node *ptrname;
 if(display == nowhere)
   ptrname = ptrwork;
 else
  {
   /* GET THE CONNECTING ADDRESS */
   if(clp_flag)
    {
     ptrname = ptrlast;
     clp_flag = FALSE;
    }
   else
    ptrname = nowhere;
  }
 if(ptrname != nowhere)
  {
   printf("\n");
   fprintf(iptr, "\n");
   if(display == nowhere)
    {
     delay(DELAY_LOOP);
     printf("Working ");
     fprintf(iptr, "Working ");
    }
   printf("NODE(%d, %d)\n", ptrname->rule_num, ptrname->con_num);
   fprintf(iptr, "NODE(%d, %d)\n", ptrname->rule_num, ptrname->con_num);
   if(display == nowhere)
    {
     printf("It's condition set is...\n");
     fprintf(iptr, "It's condition set is...\n");
    }
   /*DISPLAY NODE CONDITION SET */
   while(ptrname->condition_set[i] != ENDARRAY)
    {
     letter = ptrname->condition_set[i];
     if(letter == Q_SPACE)
       letter = SPACE;
     else
     if(letter == O_PAREN)
       letter = O_PARENTHES;
```

```
      else
      if(letter == C_PAREN)
        letter = C_PARENTHES;
      printf("%c", letter);
      fprintf(iptr, "%c", letter);
      if(ptrname->condition_set[i] == C_PARENTHES)
       {
        printf("\n");
        fprintf(iptr, "\n");
       }
      +;
     }
    if(display == nowhere)
     {
      display = ptrwork;
      printf("...and connect to the following nodes: \n");
      fprintf(iptr,"...and connect to the following nodes: \n");
     }
   }
  else
   {
    printf("\nTERMINATION NODE");
    fprintf(iptr, "\nTERMINATION NODE");
   }
 }
```

```
/*****************************************************************************************
* ---------------------------------- 15.0 FinalAnalysis function --------------------------------- *
* Purpose: A FinalAnalysis function will do a simple rule base analysis from    *
*             the node connection list.                                         *
*                                                                               *
* Calling function: NONE                                                        *
*****************************************************************************************/
void FinalAnalysis(void)
{
 int dup_count, node_count;
 rule_counter = 0;
 /* DISPLAY A PROCESS MESSAGE */
 printf("\n\n********** Rulebase Structural Analysis **********\n");
 fprintf(iptr, "\n\n********** Rulebase Structural Analysis **********\n");
 /* DO THE RULE BASE ANALYSIS PROCESSING */
 while(rule_counter < total_rule)
  {
  /* GET THE ADDRESS OF THE FIRST NODE */
  display = ptrfirst;
  dup_count = node_count = 0;
  /* DO NODE SEARCHING WHILE IT IS NOT THE END OF NODE LISTING */
  while(display != nowhere)
   {
    /* CHECK IF NODE CAME FORM THE CURRENT RULE */
    if(display->rule_num == rule_counter)
     {
      /* UPDATE THE NODE COUNTER */
      node_count++;
      /* CHECK IF THIS NODE IS A DUPLICATE NODE */
      if(display->duplicate != nowhere)
          /* UPDATE THE DUPLICATE NODE COUNTER */
          dup_count++;
     }
     /* GET AN ADDRESS OF THE NEXT NODE */
    display = display->ptrnext;
   }
  /* DISPLAY AND SAVE THE RULE BASE ANALYSIS RESULT */
  printf("\nRULE NUMBER - %d", rule_counter);
  printf("\nNUMBER OF NODES - %d", node_count);
  printf("\nNUMBER OF DUPLICATE NODES - %d\n", dup_count);
  fprintf(iptr, "\nRULE NUMBER - %d", rule_counter);
  fprintf(iptr, "\nNUMBER OF NODES - %d", node_count);
  fprintf(iptr, "\nNUMBER OF DUPLICATE NODES - %d\n", dup_count);
  delay(DELAY_LOOP);
  /* UPDATE THE RULE NUMBER */
  rule_counter++;
 }
 printf("\n**************************************************\n");
 fprintf(iptr, "\n**************************************************\n");
}
```

```
/******************************************************************************************
* ------------------------ 16.0 WriteFormatToWorkingFile function -------------------------- *
* Purpose: A WriteFormatToWorkingFile function will write a block into a      *
*             given working file.  This function also checks the initialization       *
*             of a fact_base array.                                                              *
*                                                                                                      *
* Calling function: void Error(char*, int, char*);                                           *
******************************************************************************************/
char *WriteFormatToWorkingFile(int fact_type, char *fact_ptr, FILE *workfile)
{
 int block_size = 1;
 /* WRITE THE GRAPTOOL FORMAT BLOCK TO THE WORKING FILE */
 putc(fact_type, workfile);
 while(*fact_ptr != C_PARENTHES)
  {
   putc(*fact_ptr, workfile);
   block_size++;
   fact_ptr++;
  }
 putc(C_PARENTHES, workfile);
 /* CALCULATE AN APPROXIMATION SIZE OF FACT BASE */
 if(fact_type == RETRACT)
   fact_base_size = fact_base_size - block_size;
 else
 if((fact_type == ASSERT) || (fact_type == DEFFACTS))
   fact_base_size = fact_base_size + block_size;
 /* CHECK THE SIZE OF fact_base ARRAY */
 if(fact_base_size >= CON_SIZE-2)
   Error("CON_SIZE, ",CON_SIZE,", BYTES IS TOO SMALL. \n");
 return(fact_ptr);
}
```

```
/*****************************************************************************************
* ----------------------------- 17.0 FactIsInTheFactbase function ----------------------------- *
* Purpose: A FactIsInTheFactbase function will check for the existence of a     *
*             given block from the rule_base array in the fact_base array.       *
*                                                                               *
* Calling functions: 1. char *GetAFact(char*, int, char[])                      *
*                        2. int FieldUnification(char[], char[])                 *
*                        3. void ReplaceVariableWithValue(char[], char[])        *
*****************************************************************************************/
char *FactIsInTheFactbase(char fact_str[], int *found)
{
 int match = FALSE, endvar;
 char a_fact[FACT_SIZE + VARIABLE_SIZE], *match_ptr, *c_ptr;
 c_ptr = fact_base;
 endvar = strlen(variable);
 /* DO THE SEARCHING OF A GIVEN FACT IN A fact_base ARRAY */
 while((!match) && (*c_ptr != (char)ENDARRAY))
  {
   match_ptr = c_ptr;
   /* COMPARE A GIVEN FACT WITH A FACT FROM THE fact_base ARRAY */
   if(*(c_ptr = GetAFact(c_ptr, DEFFACTS, a_fact)) != (char)ENDARRAY)
     match = FieldUnification(fact_str, a_fact);
   /* CHECK match IS NOT TRUE */
   if(!match)
     variable[endvar] = ENDARRAY;
   c_ptr++;
  }
 /* ASSIGN A match VALUE TO A found VARIABLE */
 *found = match;
 return(match_ptr);
}
```

```
/*****************************************************************************************
* ----------------------------------- 17.1 FieldUnification function ------------------------------ *
* Purpose: A FieldUnification function will check for any matching between  *
*             two given blocks.  One of the blocks comes from the rule_base    *
*            array and the other block comes from the fact_base array.  This    *
*            function will compare two fields from each block at a time.  If       *
*             the field from the rule_base array block is a variable field, the      *
*             function will store that field and its values in the variable           *
*             array.                                                                                 *
*                                                                                                      *
* Calling functions: 1. int LogicalOperation(char[], char[])                          *
*                          2. char *GetAField(char*, char[], int*)                            *
*                          3. char *GetAFact(char*, int, char[])                                *
*****************************************************************************************/
int FieldUnification(char fact_str[], char con_str[])
{
 double fvalue, cvalue;
 int fnotlast, cnotlast, vnotlast, logical, i, j, var_match;
 int fcount = 0, ccount = 0, ftotal, ctotal, match, found, endread, notvar;
 char ffield[FACT_SIZE], cfield[FACT_SIZE];
 char var_str[VARIABLE_SIZE], vfield[FACT_SIZE];
 char *fendptr, *cendptr, *ptr, *fptr, *f_ptr, *vfptr, *vptr, *cptr, *c_ptr;
 /* ASSIGN THE fact_str AND con_str ADDRESSES TO fptr AND cptr VARIABLE*/
 fptr = fact_str;
 cptr = con_str;
 fcount = ccount = 0;
 /* CALCULATE THE TOTAL NUMBER OF FIELDS IN THE fact_str */
 do
  {
   fcount++;
   fptr = GetAField(fptr, ffield, &fnotlast);
  }
 while(fnotlast);
 /* CALCULATE THE TOTAL NUMBER OF FIELDS IN THE con_str */
 do
  {
   ccount++;
   cptr = GetAField(cptr, cfield, &cnotlast);
  }
 while(cnotlast);
 /* ASSIGN NUMBER OF FIELDS TO ftotal AND ctotal */
 ftotal = fcount;
 ctotal = ccount;
 /* RE-INITIALIZE THE VALUE OF  fptr, cptr, fcount, AND ccount */
 fptr = fact_str;
 cptr = con_str;
 fcount = ccount = 0;
 /* DO THE FIELD UNIFICATION PROCESS */
 do
  {
   f_ptr = fptr;
   c_ptr = cptr;
   match = logical = FALSE;
```

```
/* GET A FIELD FROM fact_str  AND STORE IT IN ffield ARRAY */
fptr = GetAField(fptr, ffield, &fnotlast);
/* GET A FIELD FROM con_str AND STORE IT IN cfield ARRAY */
cptr = GetAField(cptr, cfield, &cnotlast);
/* UPDATE THE FIELD NUMBER OF ffield AND cfield */
fcount++;
ccount++;
/* CHECK FOR THE EXISTENCE OF ANY LOGICAL OPERATORS INSIDE ffield */
if(((strchr(ffield, LOGI_NOT) != ENDARRAY) ||
      (strchr(ffield, LOGI_AND) != ENDARRAY) ||
          (strchr(ffield, LOGI_OR) != ENDARRAY)) &&
               (ffield[0] != QUOTE))
 {
  logical = TRUE;
  ptr = ffield;
  /* CHECK IF ffield IS A VARIABLE FIELD */
  if(ffield[0] == S_WILDCARD)
   /* SEARCH FOR THE BEGINNING ADDRESS OF LOGICAL FIELD IN ffield */
   ptr = strchr(ffield, LOGI_AND);
  /* DO THE LOGICAL OPERATION BETWEEN ffield AND cfield */
  match = LogicalOperation(++ptr, cfield);
  /* CHECK IF THE ADDRESS INSIDE ptr IS NOT AN ffield ADDRESS */
  if(ptr != ffield)
   /* ERASE THE SUB-LOGICAL FIELD INSIDE ffield */
   *(--ptr) = ENDARRAY;
 }
else
/* COMPARING ffield WITH cfield */
if(strcmp(ffield, cfield) == SUCCESS)
  match = TRUE;
else
 {
  /* CONVERT ffield AND cfield TO REAL NUMBERS */
  fvalue = strtod(ffield, &fendptr);
  cvalue = strtod(cfield, &cendptr);
  /* COMPARING THE REAL NUMBERS OF ffield AND cfield */
  if((fvalue == cvalue)&&(*fendptr == ENDARRAY )&&(*cendptr == ENDARRAY))
    match = TRUE;
 }
/* CHECK IF ffield IS A WILDCARD OR VARIABLE FIELD */
if((ffield[0] == S_WILDCARD)
         || ((ffield[0] == M_WILDCARD) && (ffield[1] ==  S_WILDCARD)))
 {
  /* CHECK IF ffield IS OR HAS ANY LOGICAL FIELDS */
  if(!logical)
    match = TRUE;
  /* INITIALIZE THE VALUE OF found, vptr, AND var_match */
  found = FALSE;
  vptr = variable;
  var_match = TRUE;
```

```
/* CHECK FOR THE EXISTENCE OF ffield IN THE variable ARRAY */
while((!found) && (*(vptr = GetAFact(vptr,
    DEFFACTS, var_str)) != (char)ENDARRAY))
 {
  vfptr = var_str;
  /* GET A FIELD FROM variable ARRAY */
  vfptr = GetAField(vfptr, vfield, &vnotlast);
  /* CHECK IF ffield EXISTS IN THE variable ARRAY */
  if(strcmp(ffield, vfield) == SUCCESS)
    {
     cptr = c_ptr;
     found = TRUE;
     /* CHECK FOR THE EXISTENCE OF ffield VALUES */
     /* FROM variable ARRAY INSIDE THE con_str */
     do
      {
       /* GET A FIELD FROM con_str */
       cptr = GetAField(cptr, cfield, &cnotlast);
       /* GET A FIELD FROM variable ARRAY */
       vfptr = GetAField(vfptr, vfield, &vnotlast);
       /* CHECK IF ffield VALUE EXISTS INSIDE con_str */
       if(strcmp(vfield, cfield) != SUCCESS)
         var_match = FALSE;
      }
     while((var_match) && (vnotlast));
    }
 }
/* THE ffield MATCHES THE cfield */
/*IF AND ONLY IF  BOTH match AND var_match ARE TRUE */
if(match && var_match)
  match = TRUE;
else
  match = FALSE;
/* CHECK FOR THE EXISTENCE  OF ffield IN variable ARRAY*/
if(!found)
 {
  notvar = FALSE;
  /* SET i TO CURRENT ENDING OF variable ARRAY */
  i = strlen(variable);
  /* ADD AN OPEN PARENTHESIS AND ffield TO variable ARRAY */
  strcat(variable, "(");
  strcat(variable, ffield);
  /* CHECK IF ffield IS A SINGLE-FIELD VARIABLE OR WILDCARD */
  if(ffield[0] == S_WILDCARD)
    {
     /*CHECK IF ffield IS A SINGLE-FIELD WILDCARD */
     if(ffield[1] == ENDARRAY)
       notvar = TRUE;
     endread = ccount + 1;
    }
```

```
/* CHECK IF ffield IS A MULTIFIELD VARIABLE OR WILDCARD */
if((ffield[0] == M_WILDCARD) && (ffield[1] == S_WILDCARD))
  {
   /* CHECK IF ffield IS A MULTIFIELD WILDCARD */
   if(ffield[2] == ENDARRAY)
     notvar = TRUE;
   /* CHECK IF ffield IS THE LAST FILED IN fact_str */
   if(!fnotlast)
     /* UPDATE THE endread VALUE */
     endread = ctotal + 1;
   else
    {
     f_ptr = fptr;
     /* GET THE BOUNDARY FIELD OF ffield */
     fptr = GetAField(fptr, ffield, &fnotlast);
     fptr = f_ptr;
     fnotlast = TRUE;
     /* CHECK IF BOUNDARY FIELD IS A VARIABLE FIELD OR WILDCARD */
     if((ffield[0] == S_WILDCARD) ||
          ((ffield[0] == M_WILDCARD) && (ffield[1] == S_WILDCARD)))
      {
       vptr = variable;
       found = var_match = FALSE;
       /* CHECK FOR THE EXISTENCE OF BOUNDARY FIELD IN variable ARRAY */
       while((!found) && (*(vptr = GetAFact(vptr,
                                  DEFFACTS, var_str)) != (char)ENDARRAY))
        {
         vfptr = var_str;
         /* GET A FIELD FROM variable ARRAY */
         vfptr = GetAField(vfptr, vfield, &vnotlast);
         /* CHECK IF BOUNDARY FILED EXISTS IN THE variable ARRAY */
         if(strcmp(ffield, vfield) == SUCCESS)
           {
            found = TRUE;
            /* GET THE BOUNDARY FIELD VALUES FROM variable ARRAY */
            vfptr = GetAField(vfptr, vfield, &vnotlast);
            /* CHECK IF BOUNDARY FIELD VALUE MATCH CURRENT cfield */
            if(strcmp(vfield, cfield) == SUCCESS)
              var_match = TRUE;
           }
        }
       if(var_match)
        {
         notvar = TRUE;
         endread = ccount;
        }
       else
         /* UPDATE THE endread VALUE */
         endread = ctotal - (ftotal - fcount + 1) + ccount;
     }
```

```
            else
             {
              endread = ccount;
              cptr = c_ptr;
              /* SEARCH FOR THE BOUNDARY FIELD IN THE con_str */
              do
               {
                cptr = GetAField(cptr, cfield, &cnotlast);
                if(strcmp(ffield, cfield) != SUCCESS)
                    /* UPDATE THE endread VALUE */
                    endread++;
                else
                    cptr = con_str;
               }
              while((cnotlast) && (cptr != con_str));
             }
            /* CHECK IF THE ffield VARIABLE IS BOUND TO NOTHING */
            if(endread == ccount)
              cnotlast = notvar = TRUE;
          }
        }
      cptr = c_ptr;
      /* ADD ffield VALUE TO THE variable ARRAY */
      for(j = ccount; j<endread; j++)
        {
         cptr = GetAField(cptr, cfield, &cnotlast);
         strcat(variable, " ");
         strcat(variable, cfield);
        }
      strcat(variable, ")");
      /* CHECK IF ffield IS NOT A VARIABLE FIELD */
      if(notvar)
          /* RE-SETTING THE END OF variable ARRAY */
          variable[i] = ENDARRAY;
     }
   }
  }
 while((match) && (fnotlast) && (cnotlast));
 /* CHECK fact_str IF LAST FIELD HAS NOT BEEN CHECKED */
 if((fnotlast != SUCCESS) && (match))
  {
   GetAField(fptr, ffield, &fnotlast);
   /* CHECK fact_str  IF LAST FIELD IS NOT A */
   /*MULTIFIELD WILDCARD  OR VARIABLE */
   if((ffield[0] != M_WILDCARD) || (ffield[1] != S_WILDCARD))
     match = FALSE;
  }
 /* CHECK IF fact_str OR con_str HAS NOT REACHED THE END */
 if((fnotlast != SUCCESS) || (cnotlast != SUCCESS))
   match = FALSE;
 return(match);
}
```

```
/******************************************************************************************
* -------------------------------- 17.1.1 LogicalOperation function ---------------------------- *
* Purpose: A LogicalOperation function compares a given field from a fact    *
*             block of the fact_base array with a given logical field from a        *
*            condition block of the rule_base array.                               *
*                                                                                  *
* Calling function: NONE                                                           *
******************************************************************************************/
int LogicalOperation(char ffield[], char cfield[])
{
 char subfield[FACT_SIZE], logical = ENDARRAY;
 int fptr = 0, previous = NONE, current, sptr, tptr, match;
 /* DO A FIELD  LOGICAL OPERATION */
 while(ffield[fptr] != ENDARRAY)
  {
   sptr = 0;
   /* READ A SUB-LOGICAL FIELD FROM ffield TO subfield ARRAY*/
   do
    {
     subfield[sptr] = ffield[fptr];
     sptr++;
     fptr++;
    }
   while((ffield[fptr] != LOGI_AND) &&
          (ffield[fptr] != LOGI_OR) && (ffield[fptr] != ENDARRAY));
   subfield[sptr] = ENDARRAY;
   tptr = current = FALSE;
   /* GET A LOGICAL OPERATOR FROM A subfield */
   if((subfield[0] == LOGI_AND) || (subfield[0] == LOGI_OR))
    {
     logical = subfield[0];
     tptr++;
    }
   /* CHECK FOR AN EXISTENCE OF LOGICAL NOT IN A subfield */
   if(subfield[tptr] == LOGI_NOT)
     tptr++;
   /* COMPARE subfield WITH cfield ARRAY*/
   if(strcmp(&subfield[tptr], cfield) == SUCCESS)
     current = TRUE;
   /* INVERSE A current VALUE IF subfield HAS A LOGICAL NOT */
   if((tptr != 0) && (subfield[tptr-1] == LOGI_NOT))
     current = !current;
   /* CHECK IT IS THE FIRST LOGICAL OPERATION */
   if(previous == NONE)
     match = current;
   else
   /* DO THE LOGICAL OPERATION WITH PREVIOUS FIELD */
   switch(logical)
    {
     case LOGI_AND :if(current && previous)
                      match = TRUE;
                    else
                      match = FALSE;
                   break;
     case LOGI_OR :if(current || previous)
                     match = TRUE;
                   else
                     match = FALSE;
                   break;
     default      :match = current;
    }
   /* ASSIGN A current VALUE TO A previous VARIABLE */
   previous = current;
  }
 return(match);
}
```

```
/******************************************************************************************
* ----------------------------- 18.0 CheckTheFieldSyntax function -------------------------- *
* Purpose: A CheckTheFieldSyntax function will check the syntax errors of    *
*              each field in the given fact block.                                                     *
*                                                                                                                  *
* Calling function: char *GetAField(char*, char[], int*)                                        *
******************************************************************************************/
int CheckTheFieldSyntax(char *fact_ptr)
{
 char a_field[FACT_SIZE];
 int notlast, length, i, good_fact = TRUE;
 /* CHECK THE SYNTAX OF EACH FIELD IN A FACT BLOCK */
 do
  {
   /* GET A FIELD FROM THE GIVEN FACT BLOCK */
   fact_ptr = GetAField(fact_ptr, a_field, &notlast);
   /* CHECK IT IS NOT A STRING (BEGIN AND END WITH QUOTE) */
   if(a_field[0] != QUOTE)
    {
     /* CHECK IF IT IS A WILDCARD OR VARIABLE FIELD */
     if((a_field[0] == S_WILDCARD) ||
             ((a_field[0] == M_WILDCARD) && (a_field[1] == S_WILDCARD)))
       good_fact = FALSE;
     /* CHECK good_fact VALUE IS TRUE */
     if(good_fact)
      {
       /* CALCULATE THE SIZE OF a_field ARRAY */
       length = strlen(a_field);
       /* CHECK FOR THE EXISTENCE OF ANY LOGICAL */
       /* OPERATORS OR PARENTHESES IN THE FIELD */
       for(i=0; i<length-1; i++)
         {
          if(((a_field[i] == LOGI_OR) || (a_field[i] == LOGI_AND) ||
                     (a_field[i] == ':')) && (a_field[i] != a_field[i+1]))
            good_fact = FALSE;
          else
          if((a_field[i] == LOGI_NOT) ||
                  (a_field[i] == O_PARENTHES) || (a_field[i] == C_PARENTHES))
            good_fact = FALSE;
         }
      }
    }
  }
 while(notlast);
 return(good_fact);
}
```

```
/*******************************************************************************************
* --------------------------- 19.0 ReplaceVariableWithValue function ----------------------- *
* Purpose: A ReplaceVariableWithValue function will replace all the variable     *
*             fields in the given action block with its values from a variable       *
*             array.                                                                  *
*                                                                                     *
* Calling function: char *GetAField(char*, char[], int*)                              *
*******************************************************************************************/
void ReplaceVariableWithValue(char fact_str[], char result_str[])
{
 int fnotlast = TRUE, vnotlast, notdone, length;
 char ffield[FIELD_SIZE], vfield[FIELD_SIZE], *ffptr, *fact_ptr, *vptr;
 /* INITIALIZE THE VARIABLE */
 fact_ptr = fact_str;
 strcpy(result_str, "(");
 /* DO THE VALUE REPLACEMENT PROCESSES */
 while(fnotlast)
  {
   /* GET A FIELD FROM THE fact_str ARRAY */
   fact_ptr = GetAField(fact_ptr, ffield, &fnotlast);
   /* CHECK ffield IS A VARIABLE FIELD */
   if((ffield[0] == S_WILDCARD) ||
              ((ffield[0] == M_WILDCARD) && (ffield[1] == S_WILDCARD)))
    {
     /* CHECK ffield CONTAINS ANY LOGICAL FIELDS */
     if((ffptr = strchr(ffield, LOGI_AND)) != ENDARRAY)
       *ffptr = ENDARRAY;
     vptr = variable;
     notdone = TRUE;
     /*SEARCH FOR THE VALUES OF ffield IN A variable ARRAY */
     while((notdone) && (*vptr != ENDARRAY))
      {
       vptr = GetAField(vptr, vfield, &vnotlast);
       if(strcmp(ffield, vfield) == SUCCESS)
        {
         /* ADD VALUES FROM A variable ARRAY TO THE result_str ARRAY */
         while(vnotlast)
          {
           vptr = GetAField(vptr, vfield, &vnotlast);
           strcat(result_str, vfield);
           strcat(result_str, " ");
          }
         notdone = FALSE;
        }
       /* SEARCH FOR THE BEGINNING OF VARIABLE FIELD IN A variable ARRAY */
       vptr = strchr(vptr, O_PARENTHES);
      }
    }
    else
    /* IF ffield IS NOT A VARIABLE FIELD, ADD IT TO THE result_str ARRAY */
     {
      strcat(result_str, ffield);
      strcat(result_str, " ");
     }
   }
  /* CALCULATE THE SIZE OF A result_str ARRAY */
  length  = strlen(result_str);
  /* CHECK THE EXISTENCE OF ANY FIELDS IN result_str ARRAY */
  if(length > 1)
    result_str[length-1] = (char)C_PARENTHES;
  else
    result_str[0] = ENDARRAY;
}
```

```
/*****************************************************************************************
* ------------------------------------ 20.0 GetAField function -------------------------------------- *
* Purpose: A GetAField function will get a field from a block.                    *
*                                                                                  *
* Calling function: NONE                                                           *
*****************************************************************************************/
char *GetAField(char *fact_ptr, char a_field[], int *notlast)
{
 int i=0;
 fact_ptr++;
 *notlast = FALSE;
 /* READ A FILED FROM THE GIVEN FACT TO AN a_field ARRAY */
 while((*fact_ptr != SPACE)&&(*fact_ptr != C_PARENTHES)&&(i < FIELD_SIZE-2))
  {
   a_field[i] = *fact_ptr;
   fact_ptr++;
   +;
  }
 a_field[i] = ENDARRAY;
 /* CHECK FOR THE RETURNING FIELD SIZE */
 if(i >= FIELD_SIZE-2)
   Error("FIND A FIELD TOO LARGE FOR THE FIELD ARRAY.", NONE, "\n");
 /* CHECK IT IS NOT THE LAST FIELD IN A FACT */
 if(*fact_ptr != C_PARENTHES)
   *notlast = TRUE;
 return(fact_ptr);
}
```

```
/*********************************************************************************************
* ----------------------------------- 21.0 GetAFact function ----------------------------------------- *
* Purpose: A GetAFact function will get a block from a given array.              *
*                                                                                *
* Calling function: NONE                                                         *
*********************************************************************************************/
char *GetAFact(char *array_ptr, int fact_type, char a_fact[])
{
 int i=0;
 char facttype = O_PARENTHES, endch = ENDARRAY;
/* CHECK IT IS NOT AN DEFFACTS */
 if(fact_type != DEFFACTS)
  {
   endch = (char)ENDRF;
   facttype = (char)fact_type;
  }
 /* SEARCH FOR A facttype CHARACTER IN THE GIVEN ARRAY */
 while((*array_ptr != facttype) && (*array_ptr != endch))
      array_ptr++;
 /*CHECK FOR THE EXISTENCE OF A facttype CHARACTER */
 if(*array_ptr == facttype)
  {
   if(*array_ptr == O_PARENTHES)
     array_ptr--;
   /* READ A FACT FROM THE GIVEN ARRAY TO AN a_fact ARRAY */
   do
    {
     array_ptr++;
     a_fact[i] = *array_ptr;
     +;
    }
   while((*array_ptr != C_PARENTHES) && (i < FACT_SIZE-2));
   a_fact[i] = ENDARRAY;
  }
 /*CHECK FOR THE SIZE OF RETURNING FACT */
 if(i >= FACT_SIZE-2)
   Error("FIND A FACT TOO LONG FOR THE FACT ARRAY.", NONE, "\n");
 return(array_ptr);
}
```

```
/*********************************************************************************************
* -------------------------------------- 22.0 Error function ------------------------------------------- *
* Purpose: An Error function will display the error message and then stop     *
*              Graptool software execution.                                     *
*                                                                               *
* Calling function: void Message(char*, int, char*)                             *
*********************************************************************************************/
void Error(char *first_errmess, int err_number, char *second_errmess)
{
 /* DISPLAY THE ERROR MESSAGE ON A COMPUTER SCREEN */
 Message("\nERROR...", NONE, first_errmess);
 Message(ENDARRAY, err_number, second_errmess);
 /* CLOSE ALL THE OPENED FILES */
 fcloseall();
 /* RETURN THE CONTROL TO DOS OPERATING SYSTEM */
 exit(1);
}
```

```
/*******************************************************************************************
* ------------------------------------------ 23.0 Message function ----------------------------------- *
* Purpose: A Message function will display a given message and store it in      *
*             Graptool.err if that file has been opened.                          *
*                                                                                 *
* Calling function: NONE                                                          *
*******************************************************************************************/
void Message(char *first_mess, int mess_number, char *second_mess)
{
 /* CHECK FOR AN ENDARRAY CHARACTER */
 if(*first_mess != ENDARRAY)
   printf("%s", first_mess);
 /* CHECK FOR A NEGATIVE ONE */
 if(mess_number != NONE)
   printf("%d", mess_number);
 /* CHECK FOR AN ENDARRAY CHARACTER */
 if(*second_mess != ENDARRAY)
   printf("%s", second_mess);
 /* CHECK AN ERROR FILE IS OPENED */
 if(error_file_open)
  {
   /* CHECK FOR AN ENDARRAY CHARACTER */
   if(*first_mess != ENDARRAY)
     fprintf(eptr, "%s", first_mess);
   /* CHECK FOR A NEGATIVE ONE */
   if(mess_number != NONE)
    fprintf(eptr, "%d", mess_number);
   /* CHECK FOR AN ENDARRAY CHARACTER */
   if(*second_mess != ENDARRAY)
    fprintf(eptr, "%s", second_mess);
  }
}
```