OPS-2000™

User's Manual

Version 2.1

P.O. Box 77782, San Francisco, California, 94107

WWW.SILICONVALLEYONE.COM

The United States of America

SILICON VALLEY ONE LICENSE AGREEMENT

This is a legal agreement between you (LICENSEE), the end user, and Silicon Valley One (LICENSOR). By opening this package, you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, then you must return this package and all related materials to Silicon Valley One for a refund within thirty (30) days from the date Silicon Valley One shipped this product to you.

The terms and conditions set forth on this page of the agreement shall apply to use by LICENSEE of the OPS-2000 Software Product (herein referred to as "PRODUCT")

DEFINITIONS

APPLICATION means any instruction or instructions, in source-code or object-code format (referred to as binary format), for controlling the operation of a CPU.

SOFTWARE PRODUCT means materials such as Applications, information used or interpreted by Applications, and documentation relating to the use of Applications.

OPS-2000 MATERIALS is defined as all PRODUCT documentation, all PRODUCT distribution software, all PRODUCT backups, and this license agreement.

TERM

LICENSEE may terminate its rights under this Agreement by written notice to Silicon Valley One certifying that LICENSEE has discontinued use of and returned or destroyed all copies of PRODUCT.

If LICENSEE fails to fulfill one or more of its obligations under this Agreement, Silicon Valley One may, upon its election and in addition to any other remedies that it may have, at any time terminate all of the rights granted by it hereunder by not less that two (2) months written notice to LICENSEE specifying any such breach, unless within the period of such notice all breaches specified herein shall have been remedied; upon such termination LICENSEE shall immediately discontinue use and return all copies of PRODUCT subject to this Agreement.

SOFTWARE LICENSE

This software is protected by both United States copyright law and international treaty provisions. This program and all related materials are to be treated exactly as though they were a book. There is one exception, one copy of the diskettes can be made for the sole purpose of backing up your software. The running of this software, analogous to the reading of a book, can only be occurring on at most one computer at any given instance in time.

You cannot transfer this license unless it is done so permanently and with all OPS-2000 Materials. You may transfer this license permanently by embedding OPS-2000 in an Application and permanently transferring that Application to a single party accompanied by all OPS-2000 Materials. When transferring this license you must notify Silicon Valley One within thirty days of the date of transfer, and you must indicate the name and address of the party to whom this software is transferred.

You may not disassemble or reverse engineer this software. You may not modify this software. You may not embed OPS-2000 in an Application if in any way it violates the nature of this agreement.

LIMITED WARRANTY

This program, instruction manual, and reference materials are sold "as is," without warranty as to their performance, merchantability, or fitness for any particular purpose. The entire risk as to the results and performance of this program is assumed by you.

However, to the original purchaser only, Silicon Valley One warrants the magnetic diskette on which the program is recorded to be free from defects in materials and faulty workmanship under normal use for a period of thirty days from the date of purchase. If during this thirty-day period the diskette should become defective, it may be returned to Silicon Valley One for a replacement without charge.

Your sole and exclusive remedy in the event of a defect is expressly limited to replacement of the diskette as provided above. If failure of a diskette has resulted from accident or abuse, Silicon Valley One shall have no responsibility to replace the diskette under the terms of this limited warranty.

Any implied warranties relating to the diskette, including any implied warranties of merchantability and fitness for a particular purpose, are limited to a period of thirty days from date of purchase. Silicon Valley One shall not be liable for indirect, special, or consequential damages resulting from the use of this product. Some states do not allow the exclusion or limitation of incidental or consequential damages, or the above limitations might not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Table of Contents

Introduction

This user's manual is intended to serve as an introduction to the OPS-2000 system. It covers the basic concepts behind the system, and also gives some example programs.

Overview

OPS-2000 is a hybrid, interactive, rule and object based, software development environment.

The rule-based interpreter provides facilities for both forward and backward chaining. Where some languages half-support either forward or backward chaining, OPS-2000 has both types of chaining built into its design.

The small C++ interpreter contains about 75% of the features of the C++ version 1.0 language specification. A full interface to external compilers is provided so that compiled functions can be directly called from the C++ interpreter.

OPS-2000 compiles all code into an intermediate representation that it interprets.

Innovation

The OPS-2000 system was the first knowledge engineering tool in the world to be designed for the C++ programming language, it was the first commercially available C++ interpreter of any kind, and it was the first C++ product for Microsoft Windows. However, OPS-2000 v2.1's C++ interpreter is still a limited implementation of the C++ language. An OPS-2000 rule's action is a special C++ compound statement that includes language extensions to support knowledge engineering: assert statement, refute statement, printout statement, …

Creation

The OPS-2000 system was created and documented by Frank Lopez, who is also the founder of Silicon Valley One. Mr. Lopez received his BSCS from Purdue University in 1986, and his MSCS from the University of Illinois in 1987. His major areas of study and research have included computer engineering, operating system design, computer language design, parallel processing, theory of computation, object oriented programming, and artificial intelligence. He is a former NASA engineer, AT&T Bell Laboratories scientist, Oracle alumni, and Microsoft alumni. He is accessible via e-mail at frank@siliconvalleyone.com.

Notes

The design goal of OPS-2000 was to create a knowledge reasoning system from the point-of-view of a procedural object-oriented programming language: C++. The most obvious difference between OPS-2000 and its predecessors is that all rule variables are typed. OPS-2000 also provides the concept of active and inactive rule sets with dynamic run-time priorities and local agendas. Lastly, but most importantly, OPS-2000 was designed and developed to run on sequential and parallel architectures.

Many traditional artificial intelligence people preach that the LISP notation for rule representation is best: OPS5 and other rule-based languages follow this philosophy. However, Silicon Valley One recognizes the large amounts of time knowledge engineers spend finding software errors that could have been caught by the compiler had variable type information and fact format information been specified. Silicon Valley One also recognizes that OPS5-like languages tend to have knowledge reasoning and representation models that are too limited, making them ill-suited for developing complex systems.

Finally, for those who enjoy the LISP style of rule representation, we suggest taking a simple expert system problem and code it in a LISP-like rule-based language and in OPS-2000. For the most part you will find that those variable names that you mistyped, those fact formats that were missing a field, and those data fields that you had in the wrong order, will all be caught by the OPS-2000 compiler. In addition, you should find OPS-2000's interface to external compilers to be one of the best in the industry.

Silicon Valley One is committed to bringing the state-of-the-art in expert systems and object-oriented technologies to the people. Silicon Valley One welcomes your comments. Correspondence should be sent to the address given below.

```
Silicon Valley One
Attn: OPS-2000 Users Group
P.O. Box 77782
San Francisco, CA   94107
```

Manual Notation

This manual uses various types of notation. Symbols are used to describe a command's syntax. The table given below describes these symbols.

Symbol	Use
symbol	The bold font indicates a literal value that should be entered actually as it appears.
< item >	Angle brackets indicates that you must enter the enclosed item's value.
symbol | symbol	The pipe '|' serves as an associative "or" operation. This indicates that exactly one of the or's values is to be entered.
[symbol]	Square brackets indicates that the symbol's value is optional.
symbol*	The asterisk indicates that zero or more of the symbol's values is to be entered.
symbol+	The plus indicates that one or more of the symbol's values is to be entered.

Tutorial: Getting Started

This tutorial will cover some of the basic features of the system.

Your first task is to startup the OPS-2000 system. This can be done by typing "ops". This places you in the OPS-2000 Operating Environment (OE). This environment is aimed to be the C++ equivalent of a Lisp interpreter. This environment has shell variables (**editor**, **prompt**, ...), local variables (C++ globals), and program definitions.

The prompt "-->" indicates that the OE is ready to accept keyboard input. In this tutorial this prompt serves as an indication to you that you should enter into the OE any information appearing immediately after it. The OE begins processing a statement only after a carriage return has been entered.

The OE treats everything entered at the prompt as a C++ statement. Each entered statement is compiled and then executed. The OE interpreter views compiled definitions from the outer most program level, which is where global variables are declared. Consequently global variables can be directly declared and manipulated from the OE prompt. In addition, the OE has a set of **shell** variables that are used by its internal functions for such values as the OE prompt string and the current editor name used by function **edit()**. Shell variables can be displayed using the **env()** function, and set using the **set()** function. A summary of the shell variables is given in **env()**'s reference manual description.

Now for our first example.

```
--> printf("Hello World!\n")
```

The system prints out "Hello World!". Function **printf()** is the actual "C" **printf()** function from the standard "C" input/output library. A semicolon is automatically inserted at the end of each entered statement.

Our next example will show some of the OE's features.

```
--> fctns()
```

Function **fctns()** lists all of the C++ function prototypes for all of the currently loaded C++ functions. Any of these functions can be directly called from the OE. These functions can either be user or system defined. Function **fctns()** is an example of a compiled "C" system function. The user can add as many functions as needed, but the user cannot specify a function prototype that is already bound to another function definition. This binding (definition) can either be bound to a compiled or to an interpreted function. This binding is completely transparent to the user.

Function **declare()** is provided so that declarations can be made at the OE prompt. Let's add three integer variables to the OE. Variables entered at the OE prompt are treated as C++ globals.

```
--> declare("int a, b, c;")
```

Function **declare()** expects to have complete C++ declarations.

Now let's give these variables some values.

```
--> a = 10; b = 20; c = a + b;
```

Let's use **printf()** to check these variables' values.

```
--> printf("a = %d; b = %d; c = %d\n", a, b, c);
```

Now let's try some loop control.

```
--> for (a = 0; a < 10; a++) printf("OPS-2000 Rules!\n");
```

This loop prints "OPS-2000 Rules!" ten times.

Now let's try some advanced expressions.

```
--> a = b * ((b > 10) ? 10 : 5);
```

This is a "C" conditional expression. The new value of variable "a" should be 200. There is a set of **print()** functions that can be used to display an object's value.

```
--> print( a )
```

The next example will first be given and then explained.

```
--> declare("real abs(real value)\
          {\
            return((value >= 0.0) ? value : -value);\
          }");
-->
--> printf("Absolute values: %f %f\n", abs( 44.0 ), abs(-55.0 ));
Absolute values: 44.0 55.0
-->
```

In the above example we used function **declare()** to enter a function definition. The back-slash character is used to continue an input line on the next screen line, and it is immediately followed by a carriage return. Function **abs()** can now be used like any other OE function. This manner of entering a definition limits its length to that of the maximum string constant length.

Another useful function is **system()**. This function takes a string parameter and treats it as a command-line input to the resident operating system.

```
--> system("vi text.ops")
```

This example executes the vi-editor with the file name "text.ops". When the command has completed execution, it returns to the OE.

For the purpose of editing, a special function has been created that does the equivalent of the above operation: **edit()**. This function calls the editor name stored in the OE's **editor** variable. The OE has a set of special variables whose values can be set using the **set()** function. The function **env()** displays these variables with their associated values.

Function **load()** is used to load a file of OPS-2000 definitions into the system, and function **list()** can be used to list out the various types of objects that have been loaded. For example:

```
--> load("xmas.ops")
--> list("eo");
```

This example assumes that program "xmas.ops" is in the current working directory. In this example, function **load()** will first load "xmas.ops" into OPS-2000. Next, function **list()** will display what expert objects are currently loaded.

There are many other string values that can be given to function **list()**. For example:

```
--> list("fctns");       //List out global function prototypes.
--> list("vars");        //List out the global variables.
```

Function **clear()** clears and resets the system. For example if an error occurs during a **load()**, it may be necessary to clear the environment of any objects that have been loaded. Function **load()** feeds a file's contents to the OPS-2000 interpreter; when an error occurs, loading terminates. However any definitions that had been successfully loaded before the error occurred will still remain in the system. Consequently, since an existing definition cannot be overwritten by a new definition, function **clear()** may be necessary to reset the system so that the file can be reloaded.

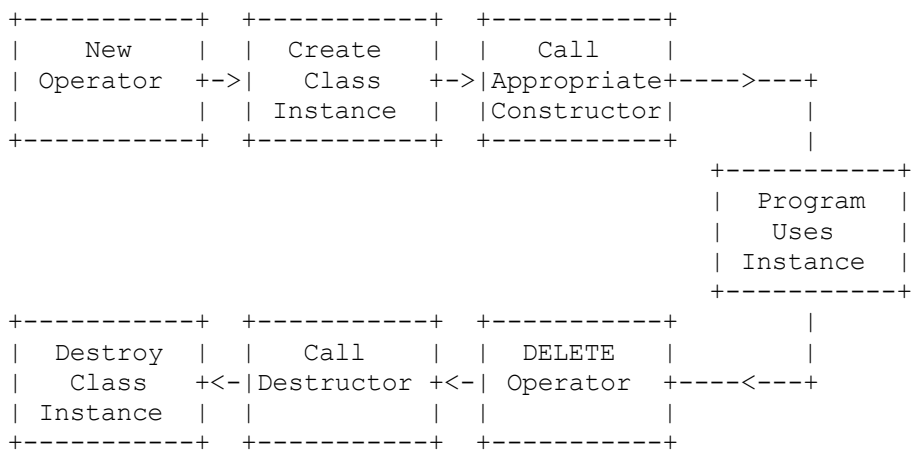
Object Oriented Programming

An object-oriented programming system is one where programs are executed by sending messages between data objects. Objects can only process messages that they have been defined to handle. In a true object-oriented system the internal structure of a data object is hidden from other objects in the system (data encapsulation). An object's internal values can only be modified by the object's operations.

Each data object has a type or class that specifies the object's data representation and the operations or messages that it can process. A class can inherit properties from other classes (inheritance).

C++ is an example of an object-oriented programming language. OPS-2000 implements an interpreted version of this language that includes all of its object-oriented features except one form of overloading. Function overloading has been implemented, but operator overloading will appear in a future release.

A C++ Class Instance's Life Cycle



The cycle given above reflects the typical class instance's (object) life cycle. A class object comes from three primary sources: it can be a **static** object in which case it is created by the system when the program begins and destroyed when the program terminates, it can be an **auto** object in which case it is created when a compound statement (block) is entered and destroyed when the block is exited, or lastly it can be a **heap** object in which case it is created by the **new** operator and destroyed by the **delete** operator. In the above diagram there are two boxes that correspond to the **new** and **delete** operators, these boxes respectively serve as the starting and finishing points of an object's life. The other two sources of class objects have starting and finishing points that can be respectively swapped for the **new** operator box and the **delete** operator box.

The use of constructors and destructors provides a way to ensure that all class objects are properly initialized and deinitialized. Improperly initialized objects tends to be a common source of software errors, so OPS-2000 has made it a requirement that each class have an explicitly declared constructor and destructor, which is not a current C++ requirement.

Expert Systems

Expert systems are computer programs that draw upon the organized expertise of one or more human experts. Human experts tend to explain how they solve problems using heuristics ("rules of thumb"). Therefore if a computer system could learn to use these same rules then it would be considered just as much an expert as any of the human experts its knowledge came from. These rules abstractly have the form "if <something is true> then <perform some actions>", and are called **productions**.

An expert system that is fabricated using a production system, codes the heuristics of an expert domain into productions. These rules are applied to the current state of the decision making process. The input to the system consists mostly of data objects that describe changes to the current state of the reasoning process. Once the system knows which rules have their conditions satisfied, it has to decide which of these to apply to the current state to generate the next state of the reasoning process. The actions associated with the selected rule(s) are executed each time a rule is applied.

By coding an expert's knowledge into productions we are able to easily update and modify any rules of the system. Furthermore, if the computer system gives a bad response, or a response that could have been better, it becomes a relatively easy task to locate the set of rules that led to the incorrect decision. As can be seen in the example OPS-2000 production given below, the heuristics the system uses have a declarative and modular representation which lends itself to the normal way humans go about problem solving.

```
//
// SaveBaby
//
defrule SaveBaby
{
       (the baby is crawling across the road)
       (a convoy of Mack Trucks is approaching)
=>
       assert("Get the baby out of danger");
}
```

This production could be a member of a set of productions that all dealt with protecting a baby from potential dangers. By giving each rule a priority, the system can determine which rule has the most certainty of giving the correct decision.

A major problem of current expert systems technology is dealing with the complexity of large systems of rules that simultaneously interact on the same problem. OPS-2000 helps deal with complexity by adding several levels of modularity to the problem solving paradigm: expert objects with interobject channels, rule sets with local agendas and name spaces, classes, and relation facts.

In OPS-2000 an expert system can be composed of many subexperts that are each an expert at a subproblem. Each of these experts can then be subdivided, and with each new subdivision the system's complexity becomes more manageable. In addition, relation facts and C++ classes both improve the management of knowledge base complexity and verification.

Production Systems

An expert system encoded as a production (rule-based) system is usually made up of three parts: a set of productions (rules) held in **production memory**, a set of data objects (assertions) held in **working memory** (blackboard/data memory/current state/knowledge base), and an interpreter (inference engine). There are two primary types of production systems: **data-driven** (forward chaining) and **goal-directed** (backward chaining). Both types of systems have many variations, including some systems that do both types of inferencing such as OPS-2000.

Data Driven

A data-driven production system uses the contents of the knowledge base to determine which productions can be executed. Therefore by having the knowledge base describe the current state of some process, the productions can then infer new states from the existing state. In an expert system these states describe the current state of the reasoning process, and the state transitions represent the application of an expert's expertise to the knowledge base. In reality we are taking a given situation and applying the rules we know pertain to it. Each time we apply a set of rules we come up with a new state. Eventually we either draw some valid conclusions, or we determine that the current state is based on incorrect assumptions and therefore is invalid. The creation of multiple current states allows one to choose the best state at any given instance of time, and thus stop the inferencing on any states that are known to be wrong. In forward chaining we say: "this is what I know, now what conclusions can I draw from this information?". So if there are quite a few possible conclusions, but only one is really necessary, this exhaustive searching may be unnecessary.

Goal Directed

A goal directed production system backward-chains the inferencing process. This is done by assuming that a goal is true and then attempting to reverse engineer the application of the heuristics of the expert domain. If all the knowledge exists in the system to support that a given asserted goal is true, then the goal becomes one of the possible solutions of the system. This can be seen as a guessing algorithm since we guess what the answer is (goal), and then attempt to prove that it is a valid solution. In reality this is equivalent to saying: "this is what I think happened, now can it be supported?" By backward chaining the new states with known data and rules we are able to rule out states that cannot be supported by the knowledge base. So if there are quite a few possible conclusions, but only one is needed, then this provides an avenue to contain the search from a fixed set of conclusions.

Components of a Production System

A production system has three main components: productions, working memory, and an interpreter.

Productions

The first component of a production system is the set of productions. A production is a condition-action construct. Each production has a name, **LHS** (Left Hand Side/antecedent/condition) and a **RHS** (Right Hand Side/consequent/action). The LHS is the logical AND of the conditional elements of the production. There are two types of condition elements usually found in the LHS of a production: tests and patterns. The test condition is usually a test for constraints on the pattern conditions that have been satisfied. A pattern condition is satisfied when it matches a data object found in the working memory. Once each element of a rule's LHS is satisfied, the rule is said to be "ready to fire". An **instantiation** is an ordered pair of a production and the working memory elements that satisfied the production's LHS. Each cycle of the production system's execution may produce one or more rules which are in the "ready to fire" state. This set of satisfied rules is called the **conflict set**, and the production system's **conflict resolution** algorithm determines which elements of the conflict set will be fired. When a rule is fired its RHS is executed and it is removed from the conflict set. This entire process is called the **recognize-act** cycle. The production system first recognizes the rules that are in the conflict set, then uses the conflict resolution algorithm to choose instantiations from the conflict set, and finally executes the RHS's of the productions associated with the selected instantiations.

In OPS-2000 there is a default recognize-act-cycle function, and there are also facilities for user defined recognize-act-cycle functions.

Working Memory

The Working Memory (WM) is a collection of data objects that represent the current state of the system. A production system's working memory is like a classroom blackboard. Each production watches the blackboard for new information. As new information (data objects) streams onto the board, a production determines all of the possible ways its LHS can be satisfied. For each possible way that the production's LHS can be satisfied, an instantiation of the production is placed into the conflict set. An instantiation is removed from the conflict set when it is fired, or if any of the working memory elements that created the instance no longer exist. The initial state of the working memory is defined by the user, and is modified using a production's RHS operators. Data objects are asserted into the working memory using an assert operator, and are retracted from the working memory using a retract operator.

Interpreter

The interpreter of a production system cycles through the recognize-act cycle. The interpreter must first match the contents of the working memory to each rule's LHS. The interpreter creates the conflict set by finding all of the productions that have become instantiated from the current working memory. Once all of the matching has been done, and a conflict set created, the interpreter uses the conflict resolution algorithm to determine the set of instantiations that it will fire. Once these instantiations have been fired, the interpreter repeats the cycle. System execution terminates when, after the match phase, the conflict set is empty. This is true because only the RHS of an instantiated production can modify the working memory.

OPS-2000 primer

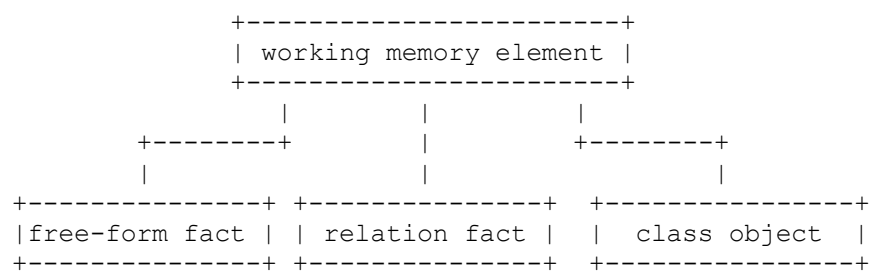
Complexity

There are four types of knowledge reasoning objects: facts, rules, rule sets, and expert objects.

1. Facts represent the active data in a knowledge reasoning process.
2. Rules act upon facts and are grouped into sets.
3. Rule sets address rule level complexity. Rule sets provide a means to package rules in a module that based on need can be added and removed from a knowledge reasoning process.
4. Expert objects provide a means to define a knowledge reasoning process. A knowledge reasoning system is composed of one or more knowledge reasoning processes (expert objects) that can share information via interobject channels.

Working Memory Elements

In OPS-2000, working memory elements are often referred to as facts, productions as rules, and instantiations as activations. The conflict set is stored in a data structure called an **agenda**. There are three types of working memory elements: free-form fact, relation fact, and class object.



Free-Form Fact

A fact is composed of symbols (words), integers, and floating point numbers (reals). A fact has at least one element.

The basic format of a fact resembles that of a sentence: a sequence of words. However a fact cannot have any punctuation. A **free-form** fact has no predefined format constraints. For example the following are all legal OPS-2000 facts:

```
1. The monkey is at position 22
2. The car is red
3. Frank is twenty four
4. The shuttle has reached main engine cutoff
5. Hurricane Gilbert missed Houston
6. I am 99.9 percent sure of the problem
7. Charles is twenty four
8. The value is 314.0E-2
```

Relation Fact

A free-form fact is treated by the compiler as a sequence of words with no constraints on its size or field types. This may seem great at first, however in use this usually leads to confusion and hard to find software errors.

Relation facts introduce format constraints, fact properties, and the flexibility of input/output formats. Four benefits of relation facts are given below.

1. Predefined format constraints can be checked by the compiler.
2. A relation fact is only pattern matched to patterns of the same relation.
3. Predefined input/output formats increase the flexibility and consistency of a knowledge base interface.
4. A relation fact can have predefined properties.

Fact relations are declared using the **defrelation** declaration which has the generic syntax given below.

```
defrelation <name> ( [<field specifiers>] ) {

      [ <field types> ]
      [ <priority specifier> ]
      [ <property specifiers> ]

[ interface:

      [ input | output <name> = <string constant> ; ]* ]

      }
```

A relation's **generic form** has the following format:

```
( <relation name> <field specifiers> )
```

The generic form is a relation fact's default input/output format.

Binary Relation

Binary relations can have zero or more of the following properties: reflexive, transitive, and symmetric. For example:

```
defrelation brother_of (?brother1 ?brother2) {
      symbol brother1, brother2;

      property = transitive;
      }

1. brother_of eddie jimmy
2. brother_of jimmy michael
      implies
3. brother_of eddie michael
```

If the first two facts exist, but the third doesn't, then the system will create a rule activation that will assert the third fact.

Relation facts take the form of the relation's generic form which is defined by the sequence of variable types given in the parentheses immediately after the relation's name. When the three example facts are compiled, they will produce the following variable bindings.

```
1. brother_of eddie jimmy
: ?brother1 = eddie; ?brother2 = jimmy

2. brother_of jimmy michael
: ?brother1 = jimmy; ?brother2 = michael
```

Priority

A relation definition can include a **priority** declaration. To understand why this declaration exists one must first understand how a relation's properties are enforced. When a binary relation has a **property** declaration, system defined rule sets are created to enforce the properties that appear in the declaration. A property enforcing rule set acts identically to a user defined rule set. Consequently, a relation's properties are enforced based on when the activations of a property enforcing rule set's rules are fired. Thus a relation's **priority** declaration is actually a **priority** declaration for any system defined rule sets that are created for the relation.

The advantage of having a relation priority is that if a relation's properties are critical to the knowledge reasoning process, then the relation can be given a very high priority, and if the relation's properties are defined to help the knowledge reasoning process when no other knowledge can be applied, then the relation can be given a very low priority.

Interface

Relation facts can also be given input/output formats. For example:

```
defrelation brother_of (?brother1 ?brother2) {
      symbol brother1, brother2;
      priority = 100;
      property = transitive;

interface:
      input in1 = "?brother1 is the brother of ?brother2";
      input in2 = "?brother1 and ?brother2 are brothers";
      output out1 = "?brother1 is the brother of ?brother2";
      }
```

Here if a fact is asserted into the knowledge base that has either the format of **in1** or **in2**, then the fact is translated to **brother_of**'s generic format. For example:

```
1. Steve is the brother of David
2. Steve and David are brothers
3. brother_of Steve David
```

Given the above relation, each of these three facts are identical. This is due to the fact that both fact-1 and fact-2 are translated to the generic form, while fact-3 is already in the relation's generic form. Thus if all three are asserted at once, and fact comparison is turned on [see **ie_eo_wm_cmp()**], only the first assertion will appear in the knowledge base. A relation's input formats are global and can only be defined once. Input formats are compared based on their variables' types and not their variables' names. When defining an input relation, each of the relation's variables must be used exactly once. The system declares a generic input/output format for each relation.

A limited form of a natural language interface can be created by declaring relations to have many possible input formats and by defining all system responses in terms of defrelation output formats. This would allow for input and output formats to be modified system-wide by simply modifying a set of relation declarations.

Class Object Fact

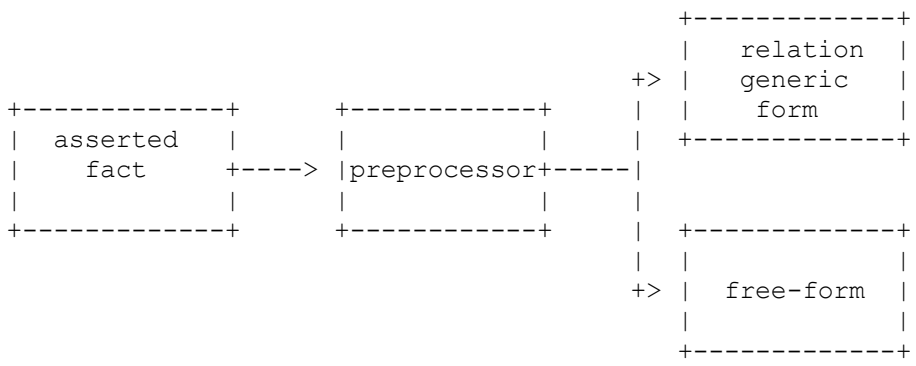
A fact can also be a C++ class object. For example:

```
class monkey {
      symbol name;            /* Monkey's name. */
      symbol on;              /* What monkey is on. */
      class object *holds;    /* Object the monkey holds. */
      class location at;      /* Location of monkey. */
      };
```

An instance of class monkey can be asserted into the knowledge base. A class object fact does not make a copy of the asserted class object, but rather it contains a reference to its asserted class object. Consequently, a class object can have multiple references appearing in the knowledge base and in the local C++ interpreter environment. Therefore some special attention must be given to the retraction of a class object from the knowledge base. A **retract** statement can be used to remove a class object from the knowledge base. However if the real intent is to delete the class object from OPS-2000, then a **delete** followed by a **retract** is required. For more information, see **retract** in the reference manual.

Fact Preprocessor

Any time a nonclass fact is asserted it is preprocessed to determine whether or not it is a relation or free-form fact. A relation fact is translated into its generic form.



Patterns

Each of the three WME types have a corresponding pattern type. A free-form fact is only compared to free-form fact patterns. A relation fact is only compared to patterns of its relation's type. Lastly, a class object is only compared to patterns of the object's class.

Free-Form Fact

Fact patterns are used to match free-form facts. The generic syntax is:

```
( <pattern element>+ )
```

The below table gives some examples of free-form fact pattern matching.

```
real r_val1;
```

fact pattern	fact object	matches
(Houston is in Texas)	(Houston is in Texas)	yes
(houston is in texas)	(Houston is in Texas)	no
(life is_a dream)	(life is yesterday)	no
(integer 1 real ?r_val1)	(integer 1 real 90.0)	yes ?r_val1 = 90.0
(1.0 2.0 3.0)	(1 2 3)	no
(1.0 2.0 3.0)	(1.0 2.0 3)	no

Relation Fact

Relation patterns are used to match relation facts. The generic syntax is:

(<relation name> <pattern element>+ **)**

A relation pattern's form must match in type and quantity that of the relation's generic form.

The below table gives some examples of relation fact pattern matching.

```
defrelation product (?name ?price ?quantity) {
      symbol name;
      real price;
      integer quantity;

interface:
      input in1 = "?name sells for ?price and we have ?quantity";
      input in2 = "?name ?quantity ?price";
}

real r_val1;
integer i_val1;
```

relation pattern	relation fact	matches
(apple ?r_val1 ?i_val1)	(apple 0.26 125)	yes ?r_val1 = 0.26 ?i_val1 = 125
(apple ?r_val1 ?i_val1)	(apple 125 0.26)	yes ?r_val1 = 0.26 ?i_val1 = 125
(apple ?r_val1 ?i_val1)	(apple sells for 0.26 and we have 125)	yes ?r_val1 = 0.26 ?i_val1 = 125
(apple ?r_val1 ?i_val1)	(apple sells for 125 and we have 0.26)	no

Class Object Fact

Class patterns are used to match class object facts.

{<class name> [<member-attribute> <pattern | pattern_element>]* **}**

A class pattern can contain zero or more member attribute specifications. The member attribute can be either matched to a pattern element or to a class pattern. If it is a class pattern, then the member attribute must be a pointer to the nested pattern's class.

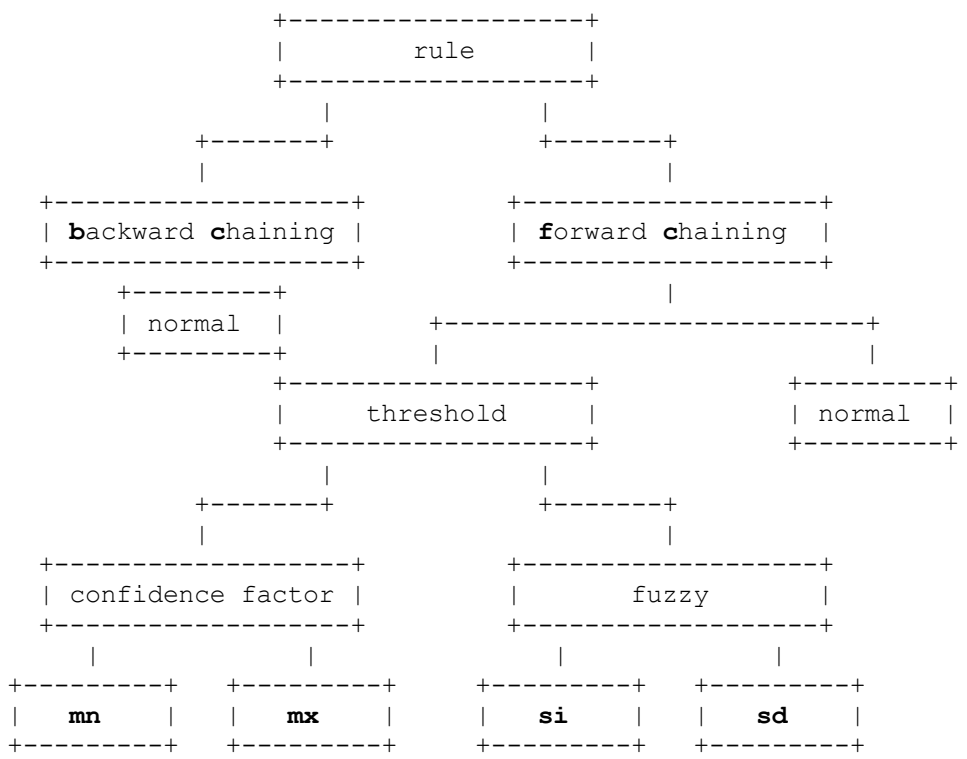
The following C++ class will be used in the examples given below.

```
class bread {
      symbol name;
      real price;
      integer quantity;
};

symbol sym1;

class pattern                  class object            matches
============================================================
{bread ^name ?sym1}            {bread ^name wonder}    yes
                                                       ?sym1 = wonder

{bread ^name country}          {bread ^name country}   yes

{bread}                        {bread ^name generic}   yes
```

A class pattern without any class attributes will match all class objects of that type.

Rules

There are two primary types of rules in OPS-2000: forward chaining and backward chaining. Forward chaining rules can be further divided into two more groups: those having threshold expressions, and those that don't. A rule type summary diagram is given below.



Forward Chaining

A forward chaining rule has the generic syntax given below:

```
defrule <name>
{
      <LHS>
=>
      <RHS>
}
```

A rule's LHS consists of pattern logic. When a rule's LHS is satisfied (matched) an activation of the rule is placed into the agenda. The conflict resolution algorithm decides which agenda entry is to be fired. When a rule is fired, its RHS is executed. This can cause assertions and retractions on the knowledge base. For example:

```
//
// RuleOne
//
defrule RuleOne
{
      (The monkey is at position 22)
=>
      assert("Move monkey to position 23");
}
```

This is a simple rule, but it shows the declarative nature of OPS-2000 rules: if the monkey is at position 22, then issue a command to move the monkey to position 23. Of course this is just our interpretation of what we visually see, the actual interpretation of the rule must be done by the knowledge reasoning system. However if knowledge base complexity is kept in check, this approach to problem solving can be very useful.

A set of logical pattern operators can be used in a forward chaining rule's pattern logic, these are: **and**, **or**, and **not**. The generic form for using these operators is:

```
( <operator> <slot specification>+ )
```

A slot specification can be a pattern condition, test condition, or pattern operator. The **not** operator checks for nonexistence and takes exactly one slot specification.

There is always an implicit logical **and** around all declared pattern logic. For example:

```
/*
 * Houston
 */
defrule Houston
{
      (houston is a city)
      (houston is in texas)
      (houston is oil)
=>
      printf("Houston is one big highway.\n");
} /*Houston*/
```

internally is represented as:

```
/*
 * Houston
 */
defrule Houston
{
      (and
            (houston is a city)
            (houston is in texas)
            (houston is oil))
=>
      printf("Houston is one big highway\n");
} /*Houston*/
```

Test conditions are for testing interpattern values.

```
/*
 * RuleOne
 */
defrule RuleOne
{
      integer x, y;                 /* rule variables */

      (Jerry is ?x years old)       //pattern condition
      (Carlos is ?y years old)      //pattern condition
      (test (?x != ?y))             //test condition
=>
      printf("Jerry and Carlos are not the same age.\n");

} /*RuleOne*/
```

Confidence Factor

```
//
// Confidence Factor
//
// mn --> min(c1, c2 ...)
// mx --> max(c1, c2 ...)
//
defrule <name> : cf [ : mn | mx ] [ :( <threshold expr> ) ]
{
      <pattern variable declarations>

      <pattern logic>

      =>

      <compound statement>
}
```

Description

Minimum

A confidence factor rule defaults to type **mn**. Variable **??** is set to the minimum confidence factor value of the match set.

Maximum

Variable **??** is set to the maximum confidence factor value of the match set. The **mx** symbol is used to specify this type.

Fuzzy

```
//
// Fuzzy rule
//
// sd --> statistically dependent.
// si --> statistically independent.
//
defrule <name> : fz [ : sd  | si ] [: ( <threshold expr> ) ]
{
      <pattern variable declarations>

      <fuzzy pattern logic>

      =>

      <compound statement>
}
```

Description

The fuzzy pattern logic can be any legal pattern logic. However the pattern logic operators **and**, **or**, and **not** take on completely new meanings.

DeMorgan's theorem is not applied to a fuzzy rule's pattern logic. The logic structure is not altered in any way.

Regardless of the logic, each and every one of the rule's test and pattern conditions must be satisfied in order for the rule to be satisfied.

Variable x is defined to be a data object matching a pattern condition: u(x) is x's degree of membership.

Variable y is the membership value of a slot specification.

Statistically Dependent

A fuzzy rule defaults to **sd**.

Its rules are:

pattern conditions	function
(<pattern condition>)	y = u(x)
(not (x))	y = 1 - u(x)
(and (x1) (x2) (x3))	y = min(u(x1), u(x2), u(x3))
(or (x1) (x2) (x3))	y = max(u(x1), u(x2), u(x3))

Statistically Independent

This is sometimes referred to as possibilistic logic.

Its rules are:

pattern conditions	function
(<pattern condition>)	y = u(x)
(not (y))	y = 1 - u(y)
(and (y1) (y2) (y3))	y = u(y1) * u(y2) * u(y3)
(or (y1) (y2) (y3))	y = (u(y1) + u(y2) + u(y3)) – (u(y1) * u(y2) * u(y3))

Backward Chaining

A backward chaining rule has the generic syntax given below:

```
defrule <name> : bc
{
      <LHS>
<=
      <RHS>
}
```

The LHS is a single goal pattern, and the RHS is a subgoal list and/or pattern logic. The pattern logic is pattern-matched exactly as though it were a forward chaining rule's LHS.

OPS-2000 has two types of working memories: fact and goal. The GWM differs from the FWM in that a goal's existence can be dependent upon the existence of other goals.

The purpose of a goal is to state a hypothesis that is to be proven or refuted.

Goals can either be of the free-form or relation data format. A goal can have one or more subgoal groups. Subgoal groups are only generated by the firing of backward chaining rules. The firing of a backward chaining rule can create at most one subgoal group. All asserted goals, regardless of their value, are entered directly into the GWM.

When a goal is asserted into a GWM it is pattern matched to the appropriate patterns. If a goal object causes a backward chaining rule to fire, then this will either prove the goal or spawn a subgoal group. Multiple rule activations can be generated by a single goal object, which when fired can generate multiple subgoal groups.

A goal can be proven in two ways: subgoal group or pattern logic. If a subgoal group has each and every member proven (retracted but not refuted), then its parent goal is proven. Pattern logic can prove a particular goal when that goal matches a backward chaining rule that has pattern logic in its RHS. In this case when the pattern logic is satisfied such that its bindings match those of the goal's LHS pattern match, then the matching goal is proven.

If a goal is proven, then it is retracted from the GWM. If a proven goal has no parent goal, then it is asserted into the FWM.

If a goal is refuted then

1. It and all of its subgoals are retracted from the GWM.
2. If it is a member of a subgoal group then that subgoal group is refuted. If that subgoal group was the last subgoal group of its parent goal and that parent goal has not matched any backward chaining rules with pattern logic, then its parent goal is refuted.

If a backward chaining rule has pattern logic, then any goal that matches the rule's goal pattern will have an anonymous subgoal group that exists for the life of the goal. This is due to the fact that the pattern logic becomes an active participant in proving the matching goal. For example:

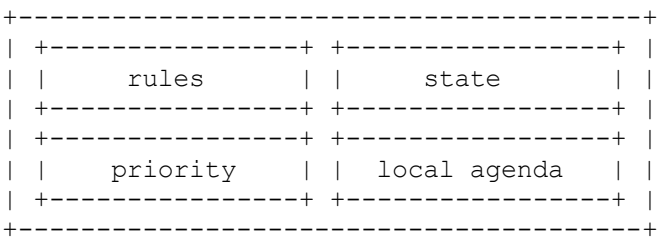
```
/*
 * Goal_Proven_by_FWM
 */
defrule Goal_Proven_by_FWM : bc
{
    segment value;

      ($?value)                    // goal pattern, matches GWM

    <=

      ($?value)                    // pattern logic, matches FWM

} /*Goal_Proven_by_FWM*/
```

This rule states that any goal that has a matching fact is proven. Consequently, any goal that matches the above goal pattern will exist until it is either proven by a subgoal group, proven by pattern logic, or is explicitly removed by the application.

Rule Sets

The purpose of a rule set is to reduce knowledge base complexity. A rule set is a collection of rules that are grouped together as a module. A rule set module can then be activated and deactivated. An activated rule set is treated as an active element of the knowledge reasoning process. A deactivated rule set is treated as though it didn't exist. When an active rule set is deactivated, its local agenda is removed from its expert object's primary agenda. Thus at runtime a rule set's state can fluctuate between being an active or inactive element of a knowledge reasoning process as the need for its knowledge varies.

A rule set is composed of one or more rules, a local state, a priority, and a local agenda. A rule set declaration cannot be nested and can only appear within an expert object's definition, making the definition local to a particular expert object.



rule set components

A rule set's state is used to control whether or not its member rules are pattern-matched to its expert object's knowledge base. If a rule set's state is active, then its rules are pattern matched. Facts are not pattern matched to an inactive rule set. A rule set's state can be given a default value by specifying a **state** declaration. This value is reset each time an expert object is reset. A rule set's runtime state can be changed using the **activate** and **deactivate** statements.

A rule set's priority is used to position its nonempty local agenda within its enclosing expert object's agenda. This priority value can be changed at runtime using the **activate** statement.

All of a rule set's activations are placed into its own local agenda. A rule set's nonempty agenda is placed in its parent expert object's primary agenda.

Local Agenda

A rule set's local agenda is composed of its rules' activations. A rule activation is placed into its rule set's local agenda based on the following rules in descending importance:

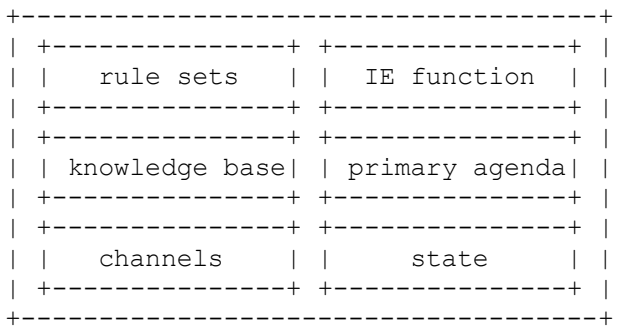
1. Highest rule activation priority.
2. Most recent working memory element.
3. Most recent time stamp.

If a rule set is active, then its nonempty agenda will be positioned in its expert object's primary agenda.

Expert Objects

An expert object is a knowledge reasoning process composed of one or more rule sets, a state, a primary agenda, an inference engine function, a knowledge base, and zero or more interobject communication channels.

The purpose of an expert object is to create a process level partition in a knowledge reasoning system's complexity. Most of today's expert system's can be encoded efficiently using a single expert object.



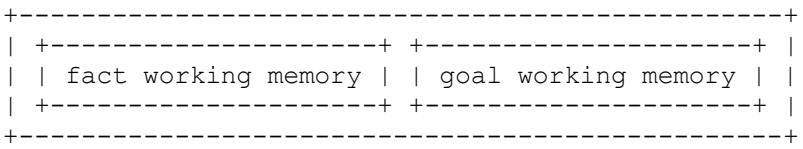
expert object components

Function **reset()** is used to reset an expert object, which sets its state to be running. The **stop** statement can be used by a rule to stop its enclosing expert object. The state flag is only meaningful if the expert object's inference engine function interprets its value. The system's default inference engine function will terminate execution when this flag is set.

The OPS-2000 operating environment provides facilities to simultaneously run multiple expert objects (running set). Information can be sent between the members of a running set via interobject communication channels.

Knowledge Base

An expert object's knowledge base has two working memories: fact and goal.



working memories

Fact Working Memory

The Fact Working Memory (**FWM**) is used to store facts for the forward chaining process. Each FWM element's existence is not directly dependent upon any other FWM element's existence.

The **deffacts** declaration is used to specify this working memory's initial state. The fact **"initial_fact"** is always asserted into the FWM when an expert object is reset.

Goal Working Memory

The Goal Working Memory (**GWM**) is used to store goals for the backward chaining process. A member of this working memory can have zero or more subgoal groups. Each subgoal group is composed of one or more goals.

The **defgoals** declaration is used to create this working memory's initial state. The goal **"initial_goal"** is always

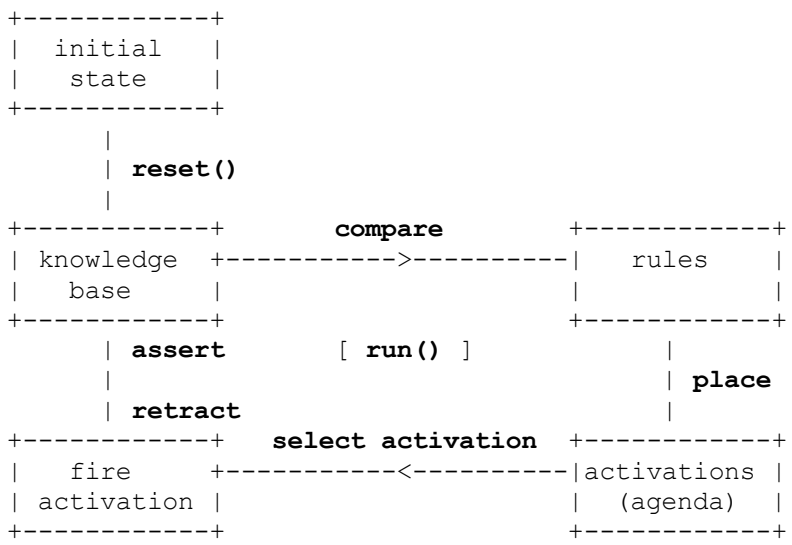
asserted into the GWM when an expert object is reset.

Inference Cycle

The inference cycle has three basic components:

1. Compare each fact to the applicable rules.
2. Place any rule activations into the agenda.
3. Select an activation to be fired; goto 1.

The knowledge base is initialized using **deffacts** and **defgoals** declarations. The facts and goals within these declarations are asserted into their respective working memories using the **reset()** function. Function **run()** is used to cycle the inference engine.



Primary Agenda

An expert object's primary agenda is composed of each of its active rule sets' nonempty local agendas. Rule sets are ordered in this agenda based on the following rules in descending importance:

1. Highest rule set priority.
2. Highest rule activation priority.
3. Most recent working memory element.
4. Most recent time stamp.

Inference Engine Function

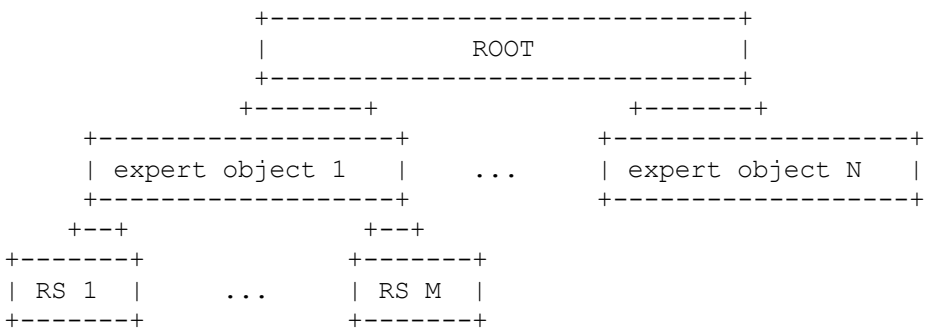
An expert object's inference engine function is responsible for selecting a rule to be fired from the primary agenda. This function can be explicitly specified for each expert object using the **ie** declaration. There is a default inference engine function for expert object's that don't have one explicitly specified. This function will fire and remove the activation that appears at the front of the primary agenda.

The default inference engine function can be directly specified using **ie_default()**. Function **ie_default()** can only be called from an externally compiled function.

There is an OPS-2000 inference engine function library that is accessible from both the interpreted and compiled environments.

Knowledge Reasoning Environment

The OPS-2000 operating environment has two primary features: the C++ interpreter and the Current Working Environment (CWE). The CWE refers to where in the knowledge reasoning system the interpreter is currently focused. The operating environment's tree-like object structure has a root with its children being all of the currently loaded expert object definitions. The root, its expert objects, and all of those expert objects' rule sets, each have their own working environments. This structure can be traversed using the **ce()** function.



This operating environment object hierarchy allows the developer to simultaneously debug and test multiple expert objects. Many functions such as **agenda()** and **facts()** give information dependent upon what the current working environment is. Variable declarations that are local to each working environment can be displayed using the **list()** function.

It is recommended that you **ce()** (change environment) to the expert object you wish to run.

A typical sequence for loading and running a particular expert object is given below.

```
-->
--> load("mab.ops")
--> ce("MFE")
--> reset()
--> run( -1 )
-->
```

MFE is the name of the expert object defined in file mab.ops.

Working Environment

The OPS-2000 interpreter has a current working environment variable. This variable is set to the object in the system with focus. A object is given focus using the **ce()** function. Function **cwe()** displays the value of the CWE variable.

Why a current working environment? Well, this allows focus to be set on a particular object so that it can be directly examined. This object can be either an expert object or a particular expert object's rule set.

For example, let's load up an example problem and show how this works. When you first start OPS-2000 the system places you at the root working environment. Anytime you want to return to this environment, from another environment, simply use the **ce()** function with no parameters. Here we use function **clear()** to remove any previously loaded definitions.

```
-->
--> ce()                  //Returns you to the root.
--> clear()         //Clears anything that has been loaded.
-->
```

Now we will load up example problem "sys.ops".

```
-->
--> load("sys.ops");
-->
```

Now the system has eight expert objects. The names of the loaded expert objects can be displayed using function **list()**.

```
-->
--> list("eo");
-->
```

In the above example, function **list()** will display the names and states of the currently loaded expert objects. To run this system we first must reset the entire system. When function **reset()** is executed at the root environment it will reset all of the loaded expert objects. It is necessary to reset a loaded expert object because when it is loaded it is initially placed in the **stopped** state. Function **reset()** places an expert object in the **running** state.

```
-->
--> reset();
-->
```

This example must be run from expert object "OutputDevice" due to its data-driven design. Running expert object "OutputDevice" can be accomplished in two ways.

```
--> run("OutputDevice", -1);   //from the root environment
--> reset();
--> ce("OutputDevice");
[OutputDevice]--> run(-1);     //from OutputDevice's environment
[OutputDevice]-->
```

The prompt string "[OutputDevice]" is the default prompt prefix when within expert object "OutputDevice". The symbol appearing between the opening and closing brackets indicates the current working environment. Function **cwe()** also will display this same string value. This CWE prefix prompt can be turned off by using the function call **set("cwe=false");**. Function **env()** can be used to display the value of the current working environment.

```
[OutputDevice]-->
[OutputDevice]--> cwe();         //Display the CWE.
[OutputDevice]--> list("rs");    //OutputDevice's rule sets.
[OutputDevice]--> list("vars"); //OutputDevice's local variables.
[OutputDevice]--> agenda();      //OutputDevice's primary agenda.
[OutputDevice]--> facts();       //OutputDevice's FWM.
[OutputDevice]--> goals();       //OutputDevice's GWM.
[OutputDevice]--> set("cwe=false");
-->                                    //Still in OutputDevice.
--> reset();                     //Reset OutputDevice.
--> run(-1);                     //Run OutputDevice.
--> ce();                              //Return to the root.
--> run(-1);                     //Run the entire system.
-->
```

Viewing Information

There is a set of system functions responsible for displaying system information. A brief summary of some of the more useful system functions is given below. Please see the reference manual for more exact information, and also note that references to a current expert object or rule set are referring to when the current working environment is set to an expert object or rule set respectively.

agenda()

Displays the agenda of the current expert object or rule set.

facts()

Displays the fact working memory of the current expert object.

goals()

Displays the goal working memory of the current expert object.

fctns()

Displays all of the loaded C++ function prototypes.

channel()

Displays a channel's contents.

list()

Lists out various types of system information: current expert object statistics, channel names, expert object names, scope, variables, rule names, and rule set names.

Monitoring Execution

OPS-2000 has a watch (trace) mode which enables a developer to verify and debug code. This watching mode enables the monitoring of function calls and returns, working memory element assertions and retractions, rule activations and deactivations, rule activation firings, inter-expert object message sending and receiving, the compiling and clearing of rules and functions, and the activities of the inference engine.

The OPS-2000 system monitoring functions are **watch()** and **unwatch()**. For example:

```
-->
--> watch("facts");      //monitor WME assertions and retractions
-->
--> watch("fctns"); //monitor calls-to and returns-from functions.
-->
--> watch("mesgs"); //watch inter-expert object message passing.
-->
```

A complete description of these functions appears in the OPS-2000 reference manual.

Dribble

dribble - to flow in an unsteady stream

The system function **goals()** displays the current value of an expert object's goal working memory. Function **goals()** is often used when debugging a backward chaining application. However sometimes it becomes necessary to record the system's output. This can be accomplished using **open_dribble()** and **close_dribble()**.

Function **open_dribble()** opens a dribble file. When a dribble file is open, the system writes all system information to the file instead of the standard output (video monitor). System information includes all output from OPS-2000 specific functions.

The Microsoft Windows version of **open_dribble()** echos all data sent to the STDIO window to the dribble file. Consequently, STDIO window data is simultaneously displayed on the window and written to the dribble file. This acts similar to the **script** command found in some operating systems.

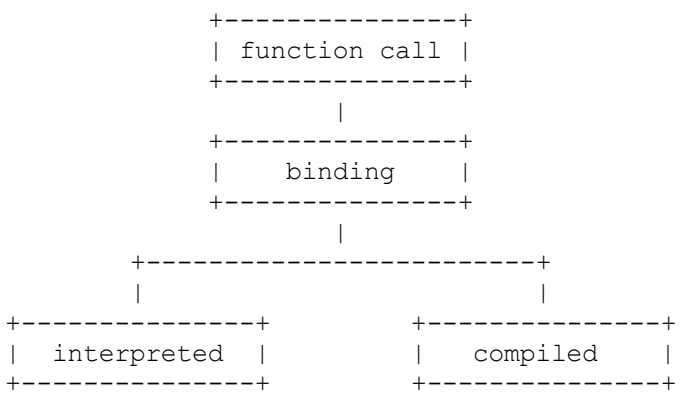
A dribble example is given below.

```
-->
--> load("dilemma.ops");             // load farmer's dilemma
--> watch("rules");                  // watch rules
--> open_dribble("one.dbl", 0);      // open dribble file: truncate
--> reset();                         // reset the system
--> run( -1 );                       // run the system until it stops
--> close_dribble();                 // close the dribble file
-->
```

In this example, all "rules" watch information is written to the dribble file. A dribble file must be closed to ensure that all dribble information has been written to the file. When using a nonWindows version of OPS-2000, this will create a file with a trace of the problem's rule firings. The rule firings are not displayed on the screen. If the problem had used the **print()** function for standard output instead of the **printf()** function, then the problem's output would also appear in the dribble file and not on the screen.

Function Calls

One of OPS-2000's best features is its ability to transparently call compiled and interpreted functions. A function call such as **fctns()** can either be bound to a compiled or interpreted function definition. A function's binding is completely transparent to the caller.



A function call is bound to a particular function definition (binding). This binding is completely transparent to the actual function call specification. There are constraints on the types of objects that can be passed-to and returned-from compiled function definitions.

Interpreted Functions

An interpreted function provides a simple and direct way to add function definitions to an OPS-2000 program.

An interpreted function can either be global (system wide use) or local to a particular class definition as are class operations, constructors, and destructors. All class function members must have interpreted bindings.

Interpreted function definitions are loaded into the system using **declare()** and **load()**. Function **declare()** has limited use when called from the interpreter which is partially due to fact that there is a system limit on the size of string constants. For example:

```
--> declare("integer x() { print(10); }")
```

The above parameter to **declare()** has a string constant parameter.

Compiled Functions

Compiled "C" functions can be directly called from OPS-2000.

There is an object file form of OPS-2000 included in the distribution disks. This file is for the host environment's "C" compiler.

The way in which OPS-2000 calls your compiled functions is through an interface defined by function **ops_def_fctn()**. This function is given a pointer to your compiled function and an associated C++ function prototype.

OPS-2000 calls function **user_fctns()** when it is started or reset. This function is not defined in the OPS-2000 object file, and must be defined by you. All of your compiled function calls to OPS-2000 functions that set some state in the system should be performed from this function. These include:

```
ie_default()
ops_def_fctn()
ops_declare()
ops_free_form()
```

The steps for adding a compiled function are outlined below.

1. Define your function and compile it into an object file for linking.
2. Place a **ops_def_fctn()** declaration for your compiled function (1) in **user_fctns()**.
3. Compile **user_fctns()** into an object file for linking.
4. Link the OPS-2000 object file with the above two object files.
5. The new executable now includes your compiled function which can be directly called from your OPS-2000 program. From the OPS-2000 system, there is no difference in the calling conventions of compiled and interpreted functions.

Embedding OPS-2000

OPS-2000 can be embedded within a larger piece of software and other systems can be embedded within it. Consequently, a particular version of OPS-2000 can be embedded within an application and simultaneously have applications embedded within it.

In terms of licensing, regardless of how OPS-2000 is embedded in an application, the application attached to OPS-2000 cannot be redistributed without an explicit **embedded** license agreement from Silicon Valley One.

Embedded Application

Applications can be embedded within OPS-2000 using the library function **ops_def_fctn()**. This compiled function-call binds an externally compiled function to an interpreted OPS-2000 C++ function prototype.

A feature of OPS-2000's external function calls is that they can pass-by-reference single dimensional arrays of data of a fundamental type (integer/int, real/double, char, symbol/char *). This allows arrays of data to be passed back and forth between OPS-2000 and external functions.

From within a rule a matching working memory element can be converted to its text representation and then subsequently passed as a string parameter. A function parameter can also be used as a buffer to retrieve data from an external source. For example:

```
/*
 * Example
 */
defrule Example
{
      match var;

      $var <- ($?)       //matches any free-form fact.

=>
      char buffer[ 120 ];

      /* Assert the matching fact into a string buffer. */
      assert( $var ) => ?buffer;

      /* Pass the buffer to an externally compiled function.*/
      printf("%s\n", ?buffer);

      /* A function can retrieve a buffer of data. */
      MyData(120, ?buffer);

      /* The character buffer can then be asserted. */
      assert(?buffer) => GWM;
      assert(?buffer) => FWM;
      assert(?buffer, Mickey, ?buffer);    //FWM

} /*Example*/
```

Embedded OPS-2000

The distribution disks contains a special form of OPS-2000 that can be embedded within another application. This special form contains a set of functions that allows an application to manipulate the OPS-2000 system. A description of these functions appears in the reference manual as the embeddable library.

An example application is given below.

```
#include <stdio.h>
#include "ops2000.h"     /*needed to run under windows*/

/*
 * OPS_2000 - Application code to be compiled and linked with the
 *            OPS-2000 embeddable object files.
 */
int OPS_2000()
{
      int i = 2;

      while (i-- > 0) {


            printf("\n=========================================\n");

            printf("\tWelcome to Indiana!!!!!\n");

            printf("\n=========================================\n");

            ops_init();                    //initialize the OPS-2000 system

            ops_load("dilemma.ops");       //load up Farmer Brown's Dilemma

            ops_ce("FarmerBrown");         //change environment to FarmerBrown

            ops_reset();                   //reset the expert object

            ops_run(-1);                   //run till stopped

            ops_terminate();               //terminate the OPS-2000 system

            printf("\n=========================================\n");
        } /*while*/


    return 0;

} /*OPS_2000*/
```

Inference Engine Library

The purpose of the inference engine library is to provide a set of functions that can be used to construct an Expert Object's inference engine function. There is an interpreter binding for each member of the inference engine library. This means an expert object's inference engine function can be either interpreted or compiled, the former providing flexibility and the latter providing speed.

Among other things, an explanation facility can be created using these functions. For example, function **ie_act_rule_info()** gives a rule's summary line as one of its return values. Thus each rule can have a declared summary line that explains its actions. The inference engine function can then either store or display these lines as each rule activation fires.

Function **ie_eo_stats()** returns an expert object's runtime statistics.

An activation can be fired multiple times before it is removed from the conflict set. This can be done using function **ie_cs_fire()** in conjunction with either **ie_cs_remove()** or **ie_cs_fire_remove()**.

Please note that using **ie_cs_fire()** on an activation followed immediately by **ie_cs_remove()** on the same activation, is not the same as using **ie_cs_fire_remove()** on that activation. This is due to the fact that when an activation fires, its position in the agenda may change. Thus **ie_cs_fire_remove()** makes sure that regardless of what happens to an activation when it fires, that it is removed from the system after it has fired.

Example Program

example file: xmas.ops

This example problem is given by creating rules that fit a story's script.

Our example is a christmas time problem. The problem is that it is a foggy christmas eve and Santa Claus cannot lift-off with his regular reindeer crew due to limited fog visibility.

The story begins with Santa pacing the floor frantically trying to think of a solution to his visibility problem. Santa, the genius he is, decides that what he needs is a headlight. So he makes a goal to find a suitable headlight for the heavy christmas eve fog.

Our example program will be a Santa expert object. So we start by declaring an expert object with the name SantaExpert. This is given below.

```
defeo SantaExpert
{

}
```

Now let's think. There are two primary components to a typical expert object definition. These being a collection of rule sets and an initial state. An initial state definition would have to encompass where the knowledge reasoning process starts. The story began: it is christmas eve and there is foggy weather.

```
defeo SantaExpert
{
      deffacts NorthPole = {
            "christmas eve",
            "foggy weather"
            }
}
```

Santa Claus, realizing he has a crisis on his hands, logs onto the north pole's computer network. Immediately recognizing the power of electronic mail (e-mail), he creates a message saying that he needs a headlight. So with the touch of a key, he sends his message across ICE-NET to all of his fellow north-pole hackers.

The following rule will do the trick.

```
      defrule SantaClaus
      {
            (christmas eve)    //If it is christmas eve
            (foggy weather)    //and foggy weather
            (not (headlight))  //and Santa doesn't have a headlight
      =>
                               //then assert a goal to
                               //get Santa a headlight.

            assert(headlight) => GWM;

      }
```

Here the LHS of the rule describes when the rule should become activated. When the rule fires, the goal "headlight" will be asserted into SantaExpert's GWM.

Luckily the Chief Helper was logged on exactly when Santa sent his message. Upon reading the message, he decides that what is needed is a bright light. So he sends a message to his subordinates requesting ideas for a bright light.

Given below is a backward chaining rule that reflects the Chief Helper's logic.

```
//
// ChiefHelper
//
defrule ChiefHelper : bc
{
       (headlight)
<=
       ((bright light))
}
```

The rule given above is a backward chaining rule so it has the type specifier "**: bc**". A backward chaining rule has a LHS goal pattern and a RHS that can be a subgoal list and/or pattern logic.

When this rule fires, a subgoal group will be spawned from the "headlight" goal which becomes the parent goal of the subgoal group. This subgoal group is specified by the list of subgoal specifications that appear between the parenthesis in this rule's RHS. Here the group only has one subgoal specification. If this subgoal group is proven, then so is the headlight goal. Likewise, since only one subgoal group will be spawned from this goal, then if this subgoal group is refuted, then so is the headlight goal.

The Candle Maker, Train Tooter, and Glitter Gluer all receive the Chief Helper's message.

The Candle Maker proposes a big candle.

```
//
// CandleMaker
//
defrule CandleMaker : bc
{
       (bright light)
<=
       ((big candle))
}
```

The Train Tooter proposes a toy train.

```
//
// TrainTooter
//
defrule TrainTooter : bc
{
       (bright light)
<=
       ((toy train))
}
```

Lastly, the Glitter Gluer proposes Rudolph's nose.

```
//
// GlitterGluer
//
defrule GlitterGluer : bc
{
      (bright light)
<=
      ((rudolphs nose))
}
```

The Chief Helper takes a look at his subordinates' recommendations, and decides that the best solution is Rudolph's nose.

First we give a rule that says if Rudolph has previously agreed that Santa can use his nose, then the goal is proven. This is to avoid asking poor Rudolph questions that he has previously answered.

```
//
// RudolphsNose
//
defrule RudolphsNose : bc
{
      priority = 100;

      (rudolphs nose)               //This is matched to the GWM.
<=
      (rudolphs nose)               //This is matched to the FWM.
}
```

The above rule has a LHS goal pattern which is matched against the GWM, and a RHS which is pattern logic. This rule says that if Rudolph has already agreed to give his services as a headlight, then a matching LHS goal is proven.

The Reindeer Keeper, anxiously waiting Santa's orders to lift-off, reads the goal asking whether Rudolph will give his services as a headlight for Santa's sleigh. In a flash of a second, he runs over to Rudolph and asks the big question.

```
//
// ReindeerKeeper
//
defrule ReindeerKeeper : fc
{
      match goal;
      priority = 99;                //Below that of rule: RudolphsNose.

      $goal <- (goal (rudolphs nose))

=>
      symbol answer;

      query("Rudolph, will you guide Santa's sleigh tonight? => ",
                                                              &?answer);
      if ((?answer == yes) || (?answer == y))
            retract $goal;
      else
            refute $goal;

} /*ReindeerKeeper*/
```

The above forward chaining rule has a LHS logic that matches the goal requesting Rudolph's nose. The RHS uses an OPS-2000 library function to ask Rudolph the question. If Rudolph says "yes" then the goal is proven (retracted), otherwise the goal is refuted.

At last Santa has his headlight, and Rudolph makes his break for fame. Santa makes one last statement which is given in the below rule.

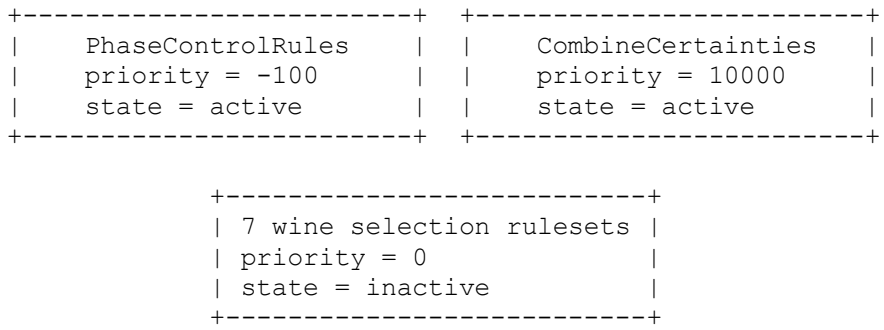
```
//
// MerryChristmas
//
defrule MerryChristmas
{
    (christmas eve)
    (foggy weather)
    (headlight)

=>
    printf("Merry Christmas! Ho! Ho! Ho!\n");
}
```

Advanced Use of Rule Sets

example file: wine.ops

The Wine Expert (WinEx) example program given in the OPS-2000 example set is a good example of the power of rule sets.

WinEx has nine rule sets. Two of which are always active: CombineCertainties and PhaseControlRules.



Rule set CombineCertainties is given the highest priority in the system, it is responsible for the combination of wine certainties. This rule set utilizes a fuzzy rule to perform this task.

Rule set PhaseControlRules is given the lowest priority in the system, its responsibility is to sequentially activate the seven rule sets responsible for the wine selection process. Each rule set is allowed to run to completion and is then subsequently deactivated by the PhaseControlRules rule set which then activates the next rule set in the process. When all of the wine selection rule sets have been run, PhaseControlRules deactivates CombineCertainties and itself, and then stops the WinEx expert object.

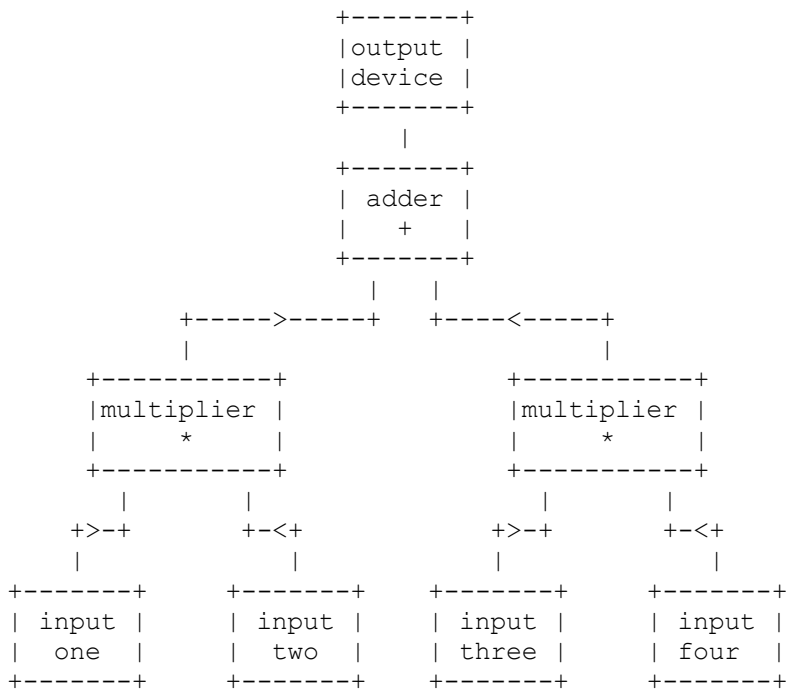
PhaseControlRules activates and deactivates the wine selection rule sets in the following order.

```
1.    activate    AskQuestions

2.    deactivate  AskQuestions
      activate    ChooseWineQualities

3.    deactivate  ChooseWineQualities
      activate    RecommendWineQualities

4.    deactivate  RecommendWineQualities
      activate    DefaultWineQualities

5.    deactivate  DefaultWineQualities
      activate    SelectWines

6.    deactivate  SelectWines
      activate    RemovePoorWineChoices

7.    deactivate  RemovePoorWineChoices
      activate    PrintWines

8.    deactivate  PrintWines
      deactivate  PhaseControlRules
      deactivate  CombineCertainties
      stop WinEx
```

Advanced Use of Expert Objects

example file: sys.ops

This example uses a set of expert objects to simulate a computer circuit. This circuit is composed of two multipliers, an adder, and an output device. The adder and the multipliers each take two inputs. The output device only takes one input. Each of the boxes in the below diagram corresponds to an expert object. The lines between the boxes represent interobject communication channels.



The input units feed input sets to the multipliers. The multipliers feed their results to the adder's inputs, and the adder sends its results to the output device.

A feature of the **receive** statement's current implementation allows for this system to be run by changing environment to that of the output-device's expert object, and then running it. The receive statement will cause demand-driven input to each of the expert objects from the four input units to the output device. When the four input units stop producing data, the multipliers will stop. Once the multipliers have stopped, the adder will stop which stops the output device.

Example Object Oriented Program

example file: pizza.ops

Adventure Game

This example program uses no public class data members. This program was modelled after a Scheme program given as a class assignment at the University of Illinois. Scheme is a Lisp dialect.

This adventure game takes place on a large midwestern campus. The game uses ten locations, three students, a dean, and a misplaced troll. As with every campus, there are things randomly distributed about it such as beers, pizzas, and magical wands, the significance of each thing will be explained below.

You may be asking yourself the question: what makes this an adventure game? Well this misplaced troll, whose name is Grendell, is really a pizza-loving cannibalistic person. There are rumors that Grendell is really a graduate student who spent one to many years writing his thesis, and consequently resorted to cannibalism to get his thesis advisor off his back. The dean, whose name is Dean, doesn't tolerate beer on campus, but otherwise is human. The students, well, are students, they study twenty four hours a day, and wander the campus pondering how to pay next semester's tuition bill. The students also pick up anything they find such as pizzas, magical wands, and beers.

Each person starts off in their natural locations: Grendell in the dungeon, Dean in the dean's office, and the students Charlene, Frank, and Karen, in Charlene's office, the Computer Laboratory, and the Altgeld (math) building respectively.

Each person has a threshold level which is a measurement of the time interval required for the person to become so bored that they have to do something. There is a system clock that all instances of people share. This clock acts like father-time, for with each tick each person becomes a bit more restless. When a person becomes so restless that their threshold level is reached, they move. When a person moves, their restlessness is reset to zero.

In this program there are three types of people: normal, troll, and dean. Each type of person has a corresponding method of responding to a move message. Consequently, the move message is declared as virtual in class person. A summary of each type of person and what happens when they move is given below. Each summary is labelled with a person type, an english description of the person's moves, and a list of moves. The list of moves has a condition for each movement, the list is checked sequentially until a move occurs.

student

Students live in their favorite working environments. When a student moves they will either pick up something at the current location, or move to another location adjacent to their current location.

1. If there are things at the current location, pick one up.
2. Otherwise randomly move from the current location.

dean

The dean lives in the dean's office, and when restless searches the campus for beers. When the dean smashes a single beer, the dean returns to the dean's office. The dean first checks a location for beers, and if there are none, performs a beer check on a random person in the room.

1. If there are any beers at this location, pick one up, smash it, and return to the dean's office.
2. If there are any people at this location, then randomly select one and perform a beer check. If the person has beer, then take one, smash it, and return to the dean's office.
3. Otherwise randomly move from the current location.

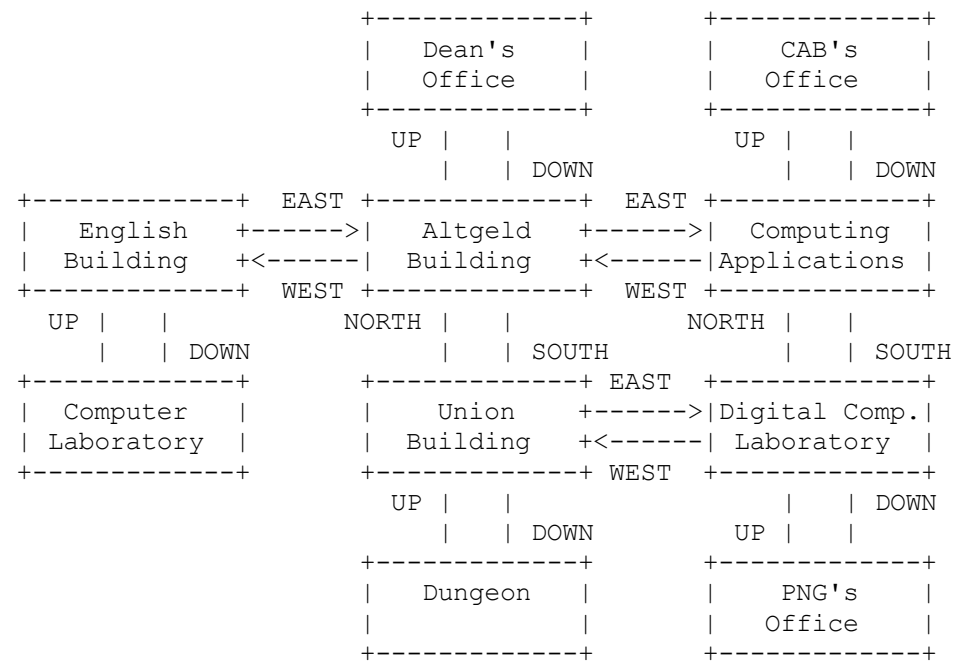
troll

The troll lives in the dungeon and eats pizzas and people. The troll prefers a good pizza over a person.

1. If there are any pizzas at this location, then pick up one, eat it, and return to the dungeon for some rest.
2. Otherwise if there are any people at this location, randomly select one to eat. If the person has a pizza, take that instead of the person's life. If the person has no pizza, allow them to pray for sometimes praying will move them to a nearby location. If the person has no pizza and praying didn't work and the person has a magical wand, then the person can wave the wand to move to a random location. However the wand only has a fixed number of charges and the random location may be the current location, so the wand won't always save a person's life. Lastly, if none of these three things have saved the person's life, eat the person and return to the dungeon for some rest.
3. Otherwise randomly move from the current location.

The Campus

The pizza adventure game uses ten campus locations. A diagram illustrating how these locations are connected is given below.



Each location understands the message "move", and responds to this message by returning a reference to the location that corresponds to the direction of the specified move. For example in the diagram given above, location "Computer Laboratory" only understands the move message "UP", and returns a reference to location "English Building" in response to this message, and returns an empty reference location to any other move message.

Each location can contain people and things. Each person has a fixed starting location, while a dozen things are randomly distributed in the ten locations. These dozen things are six pizzas, five beers, and a magical wand.

OPS-2000 Help Desk

Defrelation

Defrelation definitions are global, coming into effect as soon as they have been successfully compiled. Therefore any patterns that had been previously compiled as free-form, which matches a new relation's generic form, will still be treated as free-form even though they are now of a relation form. This is due to the fact that the pattern was compiled as free-form, and all data objects that would have matched it before the **defrelation**, will now be treated as relation data and pattern matched to a relation's patterns.

Many unrelated expert objects can be loaded into the system at once. However to avoid problems it is recommended that you load all relation definitions before any expert objects are loaded.

If an error occurs when loading a file and the file is to be reloaded again, function **clear()** has to be used if any file definitions were successfully compiled before the error occurred. This is due to the fact that any definition that is successfully loaded into the environment exists until the environment is cleared.

gensym()

A symbol created by **gensym()** exists in the system forever. Function **setgen()** provides a manner in which you can recycle **gensym()** values. For example:

```
{
      ...

       setgen( 10 );

       gensym();                //Creates symbol gen10
       gensym();                //Creates symbol gen11
       gensym();                //Creates symbol gen12

       setgen( 10 );

       gensym();                //Reuses symbol gen10
       gensym();                //Reuses symbol gen11
       gensym();                //Reuses symbol gen12

       ...
}
```

In the above example, **gensym()** is called six times, but only three symbol objects are created.

If you are constantly resetting your system, you may want to use **setgen()** in a initializing rule or function.

You should be aware of the fact that if you use **setgen()** and more than one expert system is loaded into the OPS-2000 environment, that it will affect the entire system.

Class Object Members

Resetting an expert object's knowledge base doesn't delete its class object members. These members must be explicitly deleted. Two possible methods are given below.

Method-1:

Method-1 requires that a special rule set be declared such that when it is activated, it will retract all class objects. While within an expert object environment, the following example would work:

```
--> set("rule=true");
--> activate CleanUp;
--> run( -1 );
--> reset();
```

When variable **rule** is set to **true**, the command line interpreter parses statements as if they were appearing within a rule's RHS. The **activate** statement will activate the rule set's rules which will then subsequently be pattern matched to any class objects. The rule given below would be useful in a cleanup rule set so that when all class objects have been retracted, it will **stop** the expert object.

```
defrule EndCleanUp
{
      priority = -10000; /*Lowest priority in rule set.*/

      (not (3.14))       //Some impossible fact.
=>
      stop;
} /*EndCleanUp*/
```

An example cleanup rule for class X is given below:

```
defrule CleanObject
{
      match object;
      $object <- {X}
=>
      delete $object;
      retract $object;
}
```

Method 2:

Method-2 requires that you define each class in a manner such that its constructors and destructors keep track of all asserted class objects.