OPS-2000™

Reference Manual

Version 2.1

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

WWW.SILICONVALLEYONE.COM

The United States of America

Table of Contents

Introduction

This reference manual is divided into two sections. The first section provides an overview of the OPS-2000 system. The second provides an alphabetic reference to the OPS-2000 system. This manual serves as a reference, and is not intended to be a tutorial. The OPS-2000 User's Manual should be read as an introduction to OPS-2000. A description of the predefined system functions appears at the end of this manual.

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.

Overview

This section provides an overview of some of the more general OPS-2000 concepts. The reference section should be referred to for more specific information.

The C++ interpreter is called the "Small C++ Interpreter" because currently it is not a full implementation of the C++ programming language. Future releases will enhance and improve this interpreter's capabilities. The interpreter supports: function prototypes, function name overloading, function parameter ellipsis, typedef, recursion, classes, friends, virtual, constructors, delete, destructors, for, while, do-while, if-else, switch, case, default, auto, static, integer (int), real (double), char, arrays, pointers, and nearly all of the operators with their associated precedence.

The OPS-2000 knowledge reasoning system supports: rules, rule sets, expert objects, multitasking, interobject communication, forward chaining (fuzzy/confidence factors/normal), backward chaining, inference engine control library, and three types of patterns (class/free-form/relation).

Some of OPS-2000's innovative features are:

Feature	Innovation
C++ class patterns	• Can be nested to any depth. • Attributes can be any legal C++ member expression.

Rule	• Forward and backward chaining.
	• Threshold functions
	  • Fuzzy inferencing technique
	  • Confidence factors
	• A rule's action section is a special type of C++ block.

Rule Set	• Rule names are local to their enclosing rule set. • Has its own local C++ environment. • Dynamic runtime priority capability. • Has an associated state (active/inactive).
Expert Objects	• A knowledge source composed of one or more rule sets. • Localized inference engine algorithm. • Has its own local C++ environment. • Rule set names are local to their enclosing object. • Interobject communication channels.
Parallelism	The definition of OPS-2000 will provide for future parallel processor versions. The OPS-2000 rule-based language is inherently parallel. This enables OPS-2000 code written for sequential architectures to be ported to parallel architectures with few, if any, code modifications. The internals of OPS-2000 have been designed to run on all types of computer architectures.

Notes

All keywords are lower case. There are no reserved keywords.

Four types of constants are supported: integer, real, string, and symbol.

All vectors begin with index 0.

The empty type specifier **void** is not directly supported in this current release. However there is a builtin **typedef** that defines **void** to be of type **int**.

All functions must explicitly have their return types specified.

The most obvious differences between the small C++ interpreter and a normal "C++" compiler/interpreter are: definitions can only be loaded once, there is currently no support for the "C++" preprocessor, typecasting isn't directly supported, and the **operator** declaration is not supported.

The keyword **class** must proceed all declarations specifying a class. For example:

```
class employee *x;       //Proper OPS-2000 syntax.

employee *x;             //Not supported in this release.
```

In class member functions, the applicable class object can only be directly referenced using the **this->** pointer. This pointer refers to the object that called the operation.

A forward chaining rule's RHS, like a procedure, is a C++ compound statement. This block can include a special set of statements applicable only to rules.

Memory addresses cannot be explicitly added or subtracted. A way to get the address of the 10th element of an array is to use the unary operator '&' with the array operator. For example: &x[9].

In the command line interpreter, string constants should not be manipulated in the manner shown below:

```
--> declare("char *ptr;")
--> ptr = "Hello World";
```

The last line should not be done for each statement is compiled, executed, and then completely destroyed. The destruction of a statement includes any associated string constants. Thus the value pointed to by variable **ptr** has been freed after the first statement was executed. However within a function or rule definition these statements are correct.

OPS-2000 Code Commenting

There are two methods of inserting comments into OPS-2000 source code.

Method 1: **//**

The double slash indicates to the compiler that everything from the **//** to the end of the line is to be ignored.

Method 2: **/*** ... ***/**

This method allows multiple lines to be commented. Everything from the first **/*** to the first ***/** is ignored and treated as one comment.

Word

An OPS-2000 word is a sequence of characters that begins with a character in the set {a-z, A-Z}, with the remaining characters in the set {a-z, A-Z, '_', 0-9}.

There are no reserved keywords. However one should refrain from using keywords as names (variable, type, function, ...) since this may cause some unexpected results.

Types

Fundamental

There are three primary data types: **char**, **integer**, and **real** which respectively are implemented as the "C" language's **char**, **int**, and **double**. The first two types are used to represent integers, and the last to represent floating point numbers. Type **char** is typically an 8 bit integer that can be implemented as either a signed or unsigned byte, thus for general use it really only has 7 bits of positive significance. Type **int** is typically a 32 bit integer (16 bits for IBM-PC versions). Lastly, type **real** is typically a 64 bit floating point number. A **real** number has the range 1.7E-308 to 1.7E+308 (15 digit precision).

All floating arithmetic is carried out in **double**. All integer arithmetic is carried out in **int**.

The rule based portion of OPS-2000 made it necessary to introduce a fourth data type called **symbol**. A symbol can be any OPS-2000 word. A symbol can be used anywhere in the system. It is not emphasized as a primary data type because it doesn't have a corresponding C++ fundamental type. However the small C++ interpreter fully supports this type.

Derived

From the fundamental and user defined types, other types can be derived using the declaration operators: ***** (pointer) and **[]** (vector). A pointer is the memory address of an object. A vector is a contiguous sequence of objects of a particular type.

For example:

```
integer v[ 12 ];         //Integer vector of length 12.

v[ 0 ] = 1988;           //Assignment to its 1st element
v[ 11 ] = 1999;          //Assignment to its 12th element

real probability;        //Real variable named "probability"
real *p;                 //Pointer to a real.

p = &probability;        //Variable "p" now refers to
                         //variable "probability".
```

In the last example the **&** operator takes the address of the value of the object to the right of it.

```
real matrix[10][12];     //A 10x12 vector of reals.

matrix[ 2 ];             //Refers to a vector of length 12.

p = matrix[ 1 ];         //Variable "p" now refers to the
                         //second row of matrix's object.
```

Expressions

OPS-2000 has a host of operators that are explained in the reference section of this manual. These operators include: unary, additive, multiplicative, relational, logical, conditional, assignment, and equality.

Statements

OPS-2000 supports every C++ statement except the **goto**. These are **if**, **if-else**, **while**, **do-while**, **for**, **return**, **switch**, **case**, **break**, **continue**, **default**, compound, and expression. It also has an additional set of statements that can be used with the C++ statements in a rule's action section, these are **activate**, **deactivate**, **assert**, **retract**, **refute**, **send**, **receive**, **printout**, **stop**, **excise**, and **reassert**.

The expression statement is the most commonly used statement, with the syntax:

```
<expression>;
```

The compound statement is often referred to as a **block**. It is composed of a sequence of statements that can optionally be preceded by variable declarations. Its syntax is:

```
{
[<declaration list>]
[<statement list>]
}
```

Functions

A function is a named part of a program that can be invoked from other parts of the program. Its primary components are: a return type, name, formal parameter specification, and a function body which is a compound statement. OPS-2000, unlike C++, requires that **all** functions have their return types explicitly specified.

Below is an example function to raise an integer number to an integer power.

```
/*
 * Power - the integer number x to the nth power.
 */
integer Power( integer x, integer n )
{
      integer i, value;

      value = 1;

      for ( i = 1; i <= n; i++ )
            value = value * x;

      return( value );

} /*Power*/
```

In C++ a function name can be overloaded, meaning the function call is bound based on its actual number of parameters and their respective types. Thus a function name can be used more than once provided there are no two functions with the same name and parameter descriptions. To specify a function name as being overloaded, the C++ language has the **overload** function specifier. For example:

```
      overload Power;
```

Now the function name **Power** can be given another unique formal parameter specification.

```
      /*
       * Power -- The real number x to the nth power.
       */
      real Power( real x, integer n )
      {
            integer i;
            real value;

            value = 1.0;

            for (i = 0; i <= n; i = i + 1)
                  value = value * x;

            return( value );

      } //Power
```

Classes

Classes provide a vehicle for user defined types. They provide a means for data encapsulation and guaranteed data initialization. A class definition consists of two specifications: the data needed to create an object of the type, and the set of operations for manipulating objects of the type.

A class can inherit properties from a single **base** (parent) class. The class inheritance tree is acyclic, meaning a class cannot be its own ancestor. A class with a base class is called a **derived** class.

There are two parts to a class's data specification: **private** and **public**. The private part specifies the data members that can only be accessed by a set of class defined functions (operations). The public part specifies the data members that can be manipulated by any function.

Using the **this** pointer, class functions can access the requesting object's data members. For example:

```
/*
 * employee
 */
class employee {
      //Private data member section
      integer security_level;  //Private data member

public:
      //Public data member section
      char name[ 120 ];        //Public data member
      integer id;              //Public data member
      void Print();                 //Function member
};

/*
 * Employee :: Print
 */
void employee :: Print()
{

      printf("Employee: %s\n", this->name);
      printf("Id: %d\n", this->id);
      return;

} //Print
```

In a class definition, the position of a class function member specification doesn't affect how it is treated. There are no private and public function members. These attributes only apply to data members. In addition, a class function member name is inherently overloaded. Thus the overload declaration is not necessary for function member names.

The example given below shows a call to the **Print** operation defined for class employee.

```
...

class employee snoopy;         //Employee class object
class employee *garfield;      //Pointer to class object

...                                 //Give garfield a value.
```

```
//Object snoopy requests that it be printed.
snoopy.Print();               //Print out snoopy

//Object pointed to by garfield requests that it be printed.
garfield->Print();            //Print out garfield

...
```

Lastly, a class can have data members that are shared by all objects of its type. This is accomplished by prefixing the data member with the keyword **static**. This is referred to as a static data member.

There are four general types of class functions: constructors, destructors, operations, and friends. The first two are respectively used to create and destroy a class's objects. Operations are functions that can access the public and private members of a class object, as the **Print()** function did above. A function can be declared as an operation for exactly one class.

Friends are functions that are given permission to access a particular class's private members, but are not callable as operations on the class. Consequently a function can be a friend of multiple classes, and also an operation for a single class.

Working Knowledge

Working Memory

Overview

The working memory stores the current state of knowledge during the problem-solving process. A working memory element (WME) is an asserted data object. OPS-2000 has two types of working memories: fact and goal.

Fact Working Memory

The Fact Working Memory (FWM) is used to store facts. Facts can be of any data object type.

Goal Working Memory

The Goal Working Memory (GWM) is used to store goals. A goal can be either a relation or free-form fact. A goal WME is either the product of a direct assert, or the product of a backward chaining rule.

Representation

Overview

OPS-2000 has four types of data representation: facts, relation facts, goals, and class objects. Knowledge is processed only after it has been asserted into either the fact or goal knowledge bases (working memories). Once asserted, it is then pattern matched to the appropriate patterns.

Free Form

Free-form facts are arbitrary strings of data composed of integers, reals, and symbols. The idea behind a fact is to have a data representation that closely matches its written (English, French, German, Spanish, ...) equivalent. A free form fact has no system imposed constraints on its size or field types. For example:

```
"A Zebra has black stripes"

"The monkey is at position 22"

"The automobile has gasoline"

"There is a 0.33 chance that Gilbert will hit Houston"

"Discovery has landed"
```

Relation Facts

Relation facts are facts with predefined formats and properties. A relation fact's properties can be zero or more of the following: **reflexive**, **symmetric**, and **transitive**. These properties are enforced by the system through the use of system defined, relation specific, rule sets. Relations are defined using **defrelation**. For example:

```
//Declare relation name and generic format.
defrelation brother_of (?person1 ?person2) {

      //Declare the field types: required
      symbol person1, person2;

      //Declare the relation to be transitive: optional
      property = transitive;

//Specify an interface: optional
interface:

      //Specify two input formats, in addition to generic.
      input in1 = "?person1 is the brother of ?person2";
      input in2 = "?person1 and ?person2 are brothers";

      //Specify one output format, in addition to generic.
      output out1 = "brothers ?person1 ?person2";
      }
```

If the following two facts appeared alone in a knowledge base with the above relation, then the third fact would be generated by the system.

```
1: (brother_of dexter eugene)
2: (brother_of eugene ricky)

      implies

3: (brother_of dexter ricky)
```

The **free_form()** function sets whether or not, free-form facts and patterns are accepted by the compiler. If the free-form facts flag is set to false, then a warning message will be issued each time a free-form fact or pattern is compiled by the system. The system defaults to accepting free-form objects. However they are not recommended since they are slower to process and more likely to be the source of errors (the compiler cannot check them for the correct number of fields and field types). Free-form facts descend from the early LISP based expert system tools. Relation facts are more of a procedural language approach to knowledge representation.

Goals

Goals can either be of the free-form or relation data format. A goal can have one or more sets of subgoals. Subgoals are generated by 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.

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

A goal can be proven in two ways: subgoal group and 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 goal when a particular 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 group is refuted. If that subgoal group was the last subgoal group of its parent goal, then the parent goal is refuted.

Class Objects

Class objects are fully integrated into the knowledge reasoning and C++ interpreter environments. A class object can be asserted into the FWM, but currently there is no support for class objects in the GWM. Class objects can have pointers to other objects, as well as class object data members.

Within OPS-2000 rules, class objects can be treated like frames. Patterns can be matched against a class object and any of its members that are of type class or pointer to class. There is no system limit on the nesting depth of class patterns. Thus a data member of type class can be treated like a frame slot. For example:

```
//First define an arbitrary class.
class manager {
      symbol name;
      class employee *cook;
      class collector *irs;
      class file *records;
      };

//The pattern "manager" can have three optionally nested
//patterns. A class pattern matches based on the member
//attribute specification which appears after the ^.
{manager

      ^name Ronald
      ^cook       {employee  ^name Frank ^scale ?scale}
      ^irs        {collector ^audit ?date}
      ^records    {file  ^total ?number}

      } //pattern manager
```

Certainty Factors

A certainty factor refers to the confidence with which a knowledge base object is believed. Each working memory element has a certainty factor value associated with it. This value defaults to 1.0, and it can be specified using any legal value of type **real**. A working memory element's fuzziness refers to this same exact value. This value's interpretation depends on whether it matches a pattern in a fuzzy or confidence-factor rule.

"C" Interface

Function calls to compiled "C" functions can be made directly from OPS-2000. A compiled function can return either a real, integer, char, or symbol value. Pointer values cannot be explicitly returned from a **compiled** function. Single dimensional arrays of these four types can be passed as parameters to compiled functions. Symbols returned by compiled functions are entered into the system's symbol table. A symbol returning compiled function actually returns a character string which is entered into the system's symbol table. The table given below summarizes how C++ types are passed to compiled functions.

C++ type	Passed "C" parameter type
real	double
char	char
integer	int
symbol	char *
real *	double *
char *	char *
integer *	int *
symbol *	char **

The **ops_def_fctn()** function is used to introduce a compiled function into the OPS-2000 environment. This function call should be placed in the body of **user_fctns()**, which is executed by the system each time the system is started.

There is a special object file version of OPS-2000 provided with each release. This object file should be linked with your compiled version of **user_fctns()** and any other associated compiled functions. This will create a user specific version of OPS-2000.

The **ops_def_fctn()** function prototype is:

```
ops_def_fctn( <function name>, "<C++ function prototype>")
```

Below is an example of its use with the function **printf()**.

```
ops_def_fctn(printf, "integer printf(string ...)");
```

The ellipsis in the above function prototype indicates zero or more parameters.

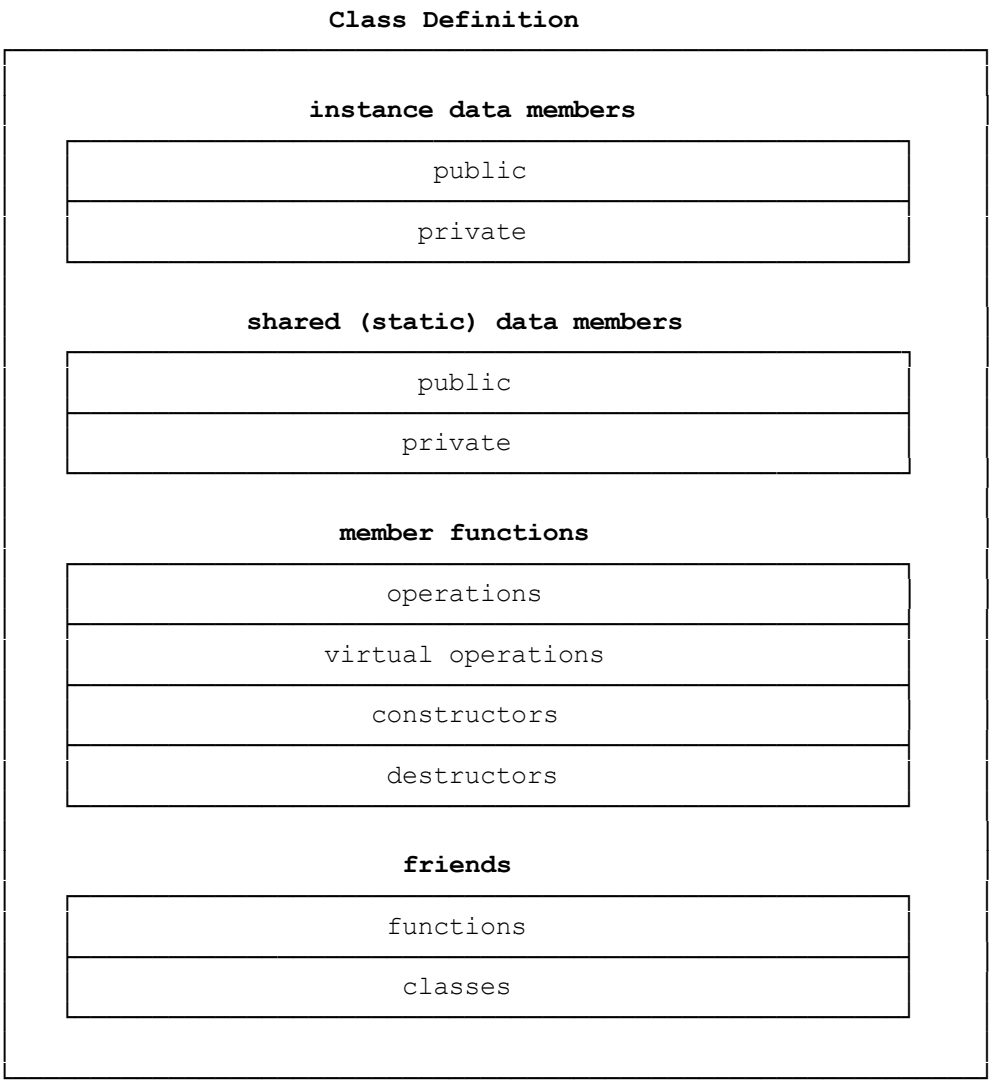
Interpreter definitions can be given to the interpreter using the **ops_declare()** function. One of the primary motives behind this particular function was to provide developers with the capability to customize their interpreter environments.

Important Note:

The "C" environment's address space is independent of the OPS-2000 address space. Consequently, addresses allocated by compiled "C" functions are local to the compiled environment, and addresses passed to the "C" environment should **NEVER** be freed by the "C" environment.

Class

Classes provide a means for new types to be created from existing types. In OPS-2000 a class definition has two required function members: a destructor, and a zero parameter constructor. The below diagram gives an overview of a C++ class definition.



The syntax and semantics of a class definition are given below.

Syntax

```
class [ <name> ] [ [ public ] : <base name> ] {

      <member list>

[ public :

      <member list>
]
}
```

The class name doesn't have to be specified if and only if it is a nested class definition. Named nested class definitions are treated as global class definitions.

The public section is optional. A class where all members are public is identical to a **struct** declaration. OPS-2000 treats a **struct** as a special form of a class declaration. Its syntax is:

```
struct [ <name> ] {

      <public member list>

}
```

Data members appearing in the member list, preceding the second **public** keyword given in the above example, are private members only accessible by member functions. Likewise, data members proceeding this keyword are accessible by all functions.

Derived

In a class definition a base class can optionally be specified. A class that includes the definition of a base class is called a **derived** class. If the **public** keyword precedes the base class name, then all public data members of the base class will also be public in the derived class.

The inheritance link for public base classes is infinite. This means that if the base class is always public, then the publics of all of the classes in the chain are available to a class declared some depth in the chain. To access a public that is from a base class or one of the classes in the chain, a class qualifier can be used.

A base class pointer can be made to point to an object of a derived class if and only if the base class is public in the derived class. For example:

```
class base *b;
class derived : public base { int a; } d;
class derived *d1;
class derived2 : public derived { int b; } d2;
class derived3 : base { int a; } d3;


b = &d;                         //Legal
d1 = &d2;                       //Legal
b = &d2;                        //Legal
b = &d3;                        //Illegal: nonpublic base class.
```

Due to the fact that OPS-2000 doesn't support explicit type casts, a nonstandard C++ assignment is supported. This assignment allows a public base class pointer to be assigned to a derived class pointer. For example:

```
d1 = b;
```

In standard C++ this would be:

```
d1 = (class derived *)b;
```

This nonstandard assignment should only be used when the base pointer being assigned was the product of a derived class's constructor. For example:

```
b = &d;
d1 = b;
```

This is required to keep a list of objects of the same base class that includes members of derived classes. For example:

```
class person {
      static class person *people;

      class person *next;

      person()
      {
            //Add the person to the list, regardless of what
            //derived type it is. To extract a person from this list
            //and assign it to its original class type such as
            //"dean" or "student" requires the above enhancement to
            //the C++ interpreter.

            this->next = this->people;
            this->people = this;
            return;

      } /*person*/

};

class dean : public person {
      ...
};

class student : public person {
      ...
};
```

Member Name Qualifier

The member name qualifier provides a means to access a member operation or public data member declared in a base class regardless of whether or not it has been redeclared in a derived class.

```
<base class name> :: <member>
```

If the base class is public, then the search for the specified member starts at that class's definition. For example:

```
class person {
public:
      char b;
      talk();
      person(); ~person();
      };

class employee : public person {
public:
      char a;
      print();
      employee(); ~employee();
    };

class manager : public employee {
public:
      char a;
      print();
      talk();
      manager(); ~manager();
    };

main()
{
      class employee e;
      class manager m;

      e.a;                          //employee::a
      e.b;                          //person::b
      m.a;                          //manager::a
      m.employee::a;                //employee::a
      m.print();                    //manager::print()
      m.employee::print();          //employee::print()
      m.employee::talk();           //person::talk()
      m.talk();                     //manager::talk()

      return;
} /*main*/
```

Description

A class member can either be a data variable declaration, function definition, or a function prototype. For example:

```
class employee {

        //A couple of private data members
        integer a[10];
        integer size;

        //A couple of function members.
        integer Print(integer, integer);
        integer Print();

        //Function member with its definition
        integer Print(real x)
        {
                printf("%f\n", x);

                for (i = 0; i < this->size; i++)
                   printf("%d\n", this->a[ i ] );

                return;
        } //Print

public :

        //Nested class definition: no name
        class {
                integer a;
                integer b;
        public:
                } no_name;

        //Nested class definition: named
        class family {
```

```
        integer total;
public :
        integer debits;
        integer credits;
        } howdy;

//A single public data member
integer x;

employee();                     //Constructor
~employee();                    //Destructor
} //Class employee
```

Data Members

Class operations access the data members of their associated class object via the pointer variable **this**. For each class operation the system defines the variable **this** to be of type pointer to the operation's class. At runtime this variable is bound by the system to the class object that is requesting the operation.

OPS-2000 currently doesn't support the C++ feature of allowing a class operation to access its associated class object's members without the use of the **this** pointer.

Private

A class's private data members come from two sources. The first source are explicit data member declarations appearing in a class definition's private section. The second source are public data members from a base class's definition. A base class's public members default to being private members in a derived class.

Only a class's function members and function friends can access the class's private data members.

A private member of a base class cannot be made public or private in a derived class.

Public

A public member of a base class can selectively be made public in a derived class's definition by using the member declaration:

```
<base class name> :: <base class member name>;
```

To access a public that is from a base class or one of the classes on the inheritance chain, a class qualifier can be used.

```
<class> :: <name>
```

These can also be used for the current class.

Static

Static class members are shared by all of the objects of the class. At runtime there is only one copy of a static data member created. Static data members are specified by prefixing the data member declaration with the keyword **static**. For example:

```
class peanuts {
      static integer a;        //shared
      integer b;

public :

      real c;
      static char message[20]; //shared
      };
```

The data members "a" and "message" are physically shared by every class peanuts object.

Visibility

At compile time, checks are made to ensure that class object members are properly accessed. Thus each time the compiler processes a class object member specification, it performs two checks.

1. Is the data member public?
2. If not (1), then is the function in which it appears declared to be a friend or member of the class?

If both (1) and (2) fail, then a compile-time error is flagged.

Function Members

In every function of a particular class X, the pointer **this** is implicitly declared as **class X *this**.

A function member's position in a class definition has no affect on how the declaration is interpreted.

There is no information hiding for member functions.

Member function names can be overloaded without the explicit use of the **overload** declaration.

There are three types of function members: operations, constructors, and destructors.

Operations

An operation is declared by specifying the operation's function prototype. When an operation's definition is declared outside of a class definition, the operation's name must be preceded by a class qualifier which takes the form:

```
<class name> :: <operation name>
```

Virtual

Virtual operations allow a programmer to declare operations in a base class that can be redefined in each derived class. The compiler will guarantee the correct correspondence between objects and their applicable operations. For example:

```
class employee {
      symbol name;
      integer id;

      /*
       *Print
       */
      virtual void Print()
      {
            printf("Name = %s; id = %d\n", this->name,
                                                this->id);
      } /*Print*/

public :
      class employee *next;

}; /*employee*/
```

The keyword **virtual** indicates that the **Print** operation can have different definitions in derived classes, and that it is the task of the compiler to find the appropriate version for each **Print** call. For example:

```
//
//manager
//
class manager : public employee {

      integer level;

      /*
       *Print
       */
      void Print()
      {

            printf("Level = %d\n", this->level);

            return;

      } /*Print*/

}; /*manager*/


//
//Demo
//
void Demo()
{
      class employee *list;

      list = new employee();
      list->next = new manager();
      list->Print();                  //employee::Print()
      list = list->next;
      list->Print();                  //manager::Print()

      return;

} /*Demo*/
```

A virtual operation must be defined for the class in which it is first declared. In addition, a virtual function cannot be a friend, constructor, or destructor function.

When a function member prototype of a derived class is identical to a virtual function member prototype of a base class, then that function member becomes a virtual operation.

Each time a function member has the keyword **virtual** prefixing its declaration, its class becomes the base class for all derived classes use of that virtual function member. Meaning a particular data object can have multiple identical virtual operations that can be applied to it. Which virtual operation to apply is determined by which base class is acting on it. For example:

```
class human {
      virtual void Print();
};

class employee : public human {
      void Print();
};

class manager : public employee {
      virtual void Print();
};

class ceo : public manager {
      void Print();
};

//
//main
//
main()
{
      class ceo charlie;
      class human *human;
      class manager *manager;

      human = &charlie;
      manager = &charlie;

      human->Print();    //Class employee's Print().
      manager->Print();  //Class ceo's Print().

      return;

} /*main*/
```

Friends

Friends are functions that have permission to access the private members of a class. A friend declaration doesn't make a function a member of the class.

There are two ways to declare a function to be a friend of a class.

```
Form 1:     friend <function prototype> ;
```

This form makes a specific function a friend of the enclosing class definition.

```
Form 2:     friend class <class name> ;
```

This form makes all function members of a specified class, friends of the enclosing class definition.

Constructors and destructors cannot be friends.

Constructor

Constructors and destructors are untyped function prototypes that have the same name as the class in which they appear. A class definition can have one or more constructors. However, a class can only have one destructor. A destructor is prefixed with a '**~**' and cannot have any declared formal parameters.

Syntax

usage

```
<class name>( <unnamed parameter list> );
```

specification

```
<class name>( <parameter list> )
                                [ : <member initializer list> ]
      <compound statement/function body>
```

Description

A constructor is called by the system to construct a class object. It can only be called using the **new** operator.

Constructor rules:

1. It cannot be explicitly called (OPS-2000 specific).
2. It cannot have a declared return type.
3. It cannot have a return statement (not enforced).
4. It cannot be a class friend.
5. If a class has a constructor with no formal parameters, then that constructor is used for objects which are not explicitly initialized. An example of this is in the declaration of arrays of class objects.

The calling order in which an object is constructed is:

1. Construct the base.
2. Construct the members of the derived.
3. Construct the derived.

Member Initializer List

The member initializer list is used to give actual parameters for member class objects' constructors. In addition, the base class constructor can have its parameters specified.

Syntax

```
            <member initializer> [ , <member initializer> ]

<member initializer> ::= [ <member name> ] ( <parameter list> )
```

Description

A member initializer without a member name is used to initialize the base class. Each given member name must correspond to a class member name of the class object (please note). Furthermore, constructors for each (argument list type/class member type) must be declared.

An example is given below:

```
class base {
      integer ct;
      base(integer in) { this->ct = in; }
      base() { ; }
      ~base() { ; }
};

class derived : base {
      derived();
      ~derived();
      derived(integer);
      class base b;
      class base d;
};

void derived :: ~derived() { ; }
void derived :: derived() { ; }
void derived :: derived(integer a)
                                : (a + 1), b(a + 2), d(a + 10)
{
      ;
}
```

Destructor

A member function of class X named ~X is called a destructor; it takes no arguments, and no return type can be specified for it. A destructor's purpose is to destroy the values of type X immediately before the object containing them is destroyed.

Syntax

```
~<class name>( <parameter list> );
```

Description

Destructor rules:

1. It cannot be explicitly called.
2. It cannot have a declared return type.
3. It cannot have a return statement (not enforced).
4. It cannot be a class friend.

The calling order in which an object is destroyed is:

1. Destroy the derived.
2. Destroy the members of the derived.
3. Destroy the base.

Constants

OPS-2000 has four types of constants: integer, real, string, and symbol. Each of these is described below.

Integer

Integer constants come in four forms: decimal, hexadecimal, octal and character. Decimal constants refer to the way you normally look at integers: 911, 928, 944, and 2000.

Hexadecimal constants are the base 16 equivalent of decimal constants. These constants have the prefixes "0x" and "0X". The integers represented by the above decimal constants respectively have the hexadecimal forms 0x38f, 0x3a0, 0x3b0, and 0x7d0.

Octal constants are the base 8 equivalents of decimal constants. These constants have the prefix "0". The above decimal constants are respectively represented by 01617, 01640, 01660, and 03720.

Character constants are eight bit integers which can either be signed or unsigned depending upon the architecture. Thus for integer values, only the range 0 to 127 can be consistently represented. Character constants have three forms: '\ddd', '\s' and 'l'. The "ddd" can be any three digit octal value. The "l" is a visible keyboard character. Lastly, the "s" is a special escape character.

The escape characters are:

```
'\b'        backspace
'\f'        form feed
'\n'        new line
'\r'        carriage return
'\t'        tab
'\v'        vertical tab
'\\'        backslash
'\''        single quote
'\"'        double quote
'\0'        integer value 0
```

Real

Real constants are used to represent floating point numbers. These constants must have exactly one floating point and no spaces. This constant has an exponential form that is specified by using an "e" or "E": <real>[**e |** E **]**<exponent>. The exponent is to the immediate right of the "e", and the floating number is to its immediate left. For example:

```
1.23            .23             1.23e10         1.23E10
0.23            1.0             1.23e-7         1.23E-7
```

String

String constants are character sequences enclosed in double quotes. For example:

```
"Hello World!"
```

Every string has one more character than it actually appears to have. This extra character is used to terminate the string and is the known as null character: '\0'. Note that '0' is not equal to '\0'. A string's type is a character vector of the appropriate length. For example "Hello" is of type char[6]. The maximum string constant length is 127 characters which includes the mandatory string termination character '\0'.

String constants are static definitions and should not be modified. For example:

```
char *greeting;

greeting = "Hello World!";

greeting[0] = 'h';
```

The above assignment is illegal, but the compiler will not flag an error.

Symbol

A symbol constant is a hashed character string that the equality operators can operate upon. A symbol is passed to compiled functions as a character string. It is delimited using the **'\'** character. For example:

```
\hello\
\this is a symbol\
```

The representation of this constant is similar to a string constant. However escape characters cannot be embedded within a declared symbol constant.

Functions

There are two ways to declare a function: usage and specification.

Usage

A function usage declaration is referred to as a **function prototype**, its purpose is to declare the function binding definition which consists of a function name, return type, and parameter types. Its general form is:

```
<return type> <name> ( <parameter types: no variable names> );
```

Specification

A function specification is similar to a function prototype, except its parameters are named and the prototype's ending ';' is replaced by a function body definition which happens to be a compound statement. However this is just the general form, for there are function specifications that can include information not specifiable with this syntax, such as free store function specifications.

defchannels

Channels are used to send messages between expert objects. A channel is a unidirectional path between two expert objects. Only one expert object can send into a particular channel and only one can receive from it. This binding of expert objects to channels is performed at compile-time.

Syntax

```
defchannels <channel name> [, <channel name> ]* ;
```

Description

Channel declarations cannot be nested within any other declaration.

Messages can only be sent and received from a rule's RHS. This is accomplished using the **send** and **receive** statements.

The current channel implementation is similar to software events. The sender of a message doesn't wait for a message to be received, for a message is sent by simply depositing it into a channel buffer. The receiver of a message doesn't wait either, but rather it must continuously poll the input channel until a message arrives. If a receiver requests a message when there are no messages, the returned input buffer will have the value of an empty string which means its first byte is set to '\0'.

If a channel has no receiver, then the buffer will just collect messages until a receiver is defined or the buffer is reset.

If a channel has no sender, then the receive statement will act as though the sender has stopped placing new messages into the buffer.

Over a channel's life-span it can have many different sending and receiving expert objects. However at any given instance in time it can have at most one sender and receiver.

defeo

An Expert Object (EO) is a knowledge source composed of zero or more rule sets with its own C++ environment. It can communicate with other EO's through channels created using **defchannels**.

There are two main themes behind an expert object: allow for communicating knowledge sources, and give the developer a way to partition the complexity of the knowledge reasoning system.

An EO's rule sets can be run in parallel.

Syntax

```
defeo <name>
{

       [ ie = <function prototype> ; ]
       [ block declarations ]*

       [ <rule set declaration> |
         <deffacts declaration> |
         <defgoals declaration>   ]*

}
```

An EO's declaration block is identical to that of a compound statement's. In terms of lexical scoping, the EO is treated as its rule sets' parent block.

Description

An expert object is a knowledge source composed of the following:

1. An inference engine function (defaults).
2. One or more rule sets.
3. Primary Agenda (PA).
4. A Fact Working Memory (FWM).
5. A Goal Working Memory (GWM).
6. A local C++ environment.
7. Zero or more channels it can send messages to.
8. Zero or more channels it can receive messages from.

The simplest expert object definition is:

```
//
//Simplest
//
defeo Simplest
{

       //
       //defrs Simplest
       //
       defrs Simplest
       {

              //
              //defrule Simplest
              //
              defrule Simplest
              {
                  (?)
              =>
                    ;
              }
       }
}
```

Inference Engine

An EO's inference engine can be explicitly specified by using the inference engine declaration syntax given below.

```
ie = <inference engine function prototype>;
```

This function must be declared with exactly two integer parameters, and it can only appear once within an expert object's definition. For example:

```
ie = integer local_ie( integer, integer );
```

The first parameter is the system's id for the expert object that called the function, this value is needed by the inference engine control library. The second parameter is the number of steps the expert object was instructed to run, this is the same value that is passed to the **run()** function, and it is the inference engine function's responsibility to interpret this value.

If this declaration does not appear, then the system uses a default inference engine function which will fire and remove at most "steps" number of activations. This default inference engine can be changed using the system function **ie_default()**.

Rule Sets

An EO is composed of one or more rule sets. These rule sets all act on the EO's working memories. Each rule set has its own local agenda to handle its rules' activations. All of the rule sets can perform their pattern matching phase in parallel, completely independent of each other. After each pattern matching cycle, the expert object orders its rule sets in its primary agenda according to the activation at the front of each of their local agendas. This ordering does not include rule sets that have empty agendas.

A rule set is placed into the primary 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.

The default inference engine will fire and then remove the activation that appears at the front of the primary agenda.

Channels

A channel is used to send messages between two EOs. This is similar to the concept of an Occam channel in that a channel is unidirectional and it can only have one sender EO and one receiver EO. However an Occam channel causes a sender to wait until a receiver is ready, and likewise a receiver must wait until a sender is ready. A sender expert object sends a message and continues running regardless of whether or not the message has been received. A receiver expert object must poll a channel until a message appears.

deffacts

Deffacts is used to describe the initial state of an EO's FWM.

Syntax

```
deffacts <name> = {
      <string constant>
      [, <string constant> ]*
      }
```

This declaration can only appear within an EO definition.

Description

An EO can have zero or more of these declarations. The facts within a deffacts definition are asserted when an EO is reset. An EO's FWM is always first initialized with the fact: "initial_fact".

defgoals

Defgoals is used to describe the initial state of an EO's GWM.

Syntax

```
defgoals <name> = {
      <string constant>
      [, <string constant> ]*
      }
```

This declaration can only appear within an EO definition.

Description

An EO can have zero or more of these declarations. The goals within a defgoals definition are asserted when an EO is reset. An EO's GWM is always first initialized with the goal: "initial_goal".

defrelation

Relations are used to define a fact's format and properties. They are particularly useful in verifying that a system excepts only predefined input formats. A system flag can be set so that only facts that match some relation's input format are accepted [see **free_form()**], any other formats will generate compile-time and run-time warning messages. Relation patterns take the form of the relation's generic format, which is:

```
( <relation name> <fields> )
```

All of a relation's fact objects are compiled into this format.

Relations must be defined before they are used.

Syntax

```
defrelation <relation name> ( [ ?<field name> ]* ) {

      [ priority = <integer constant> ; ]
      [ property = <relationship> [, <relationship> ]* ; ]*

      <field name declaration>*

[ interface :

      [ <input | output> <format name> = <fact string> ;]*
]
}

<relationship> ::= transitive | reflexive | symmetric
```

This declaration cannot be nested within any other declaration.

Description

The field list appearing within parentheses is the relation's field list. The system automatically defines an input and output format for the relation that matches the generic form. These are referred to as the generic input and output formats.

User defined input/output formats can be directly specified in a relation's optional **interface** section.

An input format must use all of a relation's field variables exactly once.

Every OPS-2000 relation input format must be unique. A variable appearing within an input format is compared and bound based solely on its type.

All field names must have a corresponding field declaration.

A relation pattern must take its relation's generic format.

A binary relation (two fields) can have one or more of the following properties: reflexive, transitive, and symmetric. These properties are declared using the **property** declaration, and are applied using system defined rule set's. The priority value of these rule sets defaults to zero. A relation's definition can explicitly specify a priority value using the **priority** declaration.

A data object that matches a relation's input format is automatically translated by the system into the relation's generic format. Therefore all patterns that are intended to match a particular relation's objects, must be of the relation's generic form. This does not mean that rule variables have to be used for each of a relation's field positions, but it does mean that all fields must be specified with the correct field type. The compiler checks to make sure that all relation patterns follow these rules.

There can be zero or more instances of a field variable in a relation's output format.

A relation's output format names can only appear after a match variable that is used within an OPS-2000 statement. The following rules summarize the use of output formats.

1. An output format can be specified by appending a format suffix to the variable. Example: $x:out1, converts $x to the string format specified by out1. The format suffix is looked up at runtime. If the format doesn't exist in the fact's relation, then the suffix is ignored and (2) is applied.
2. If no relation format is specified, then the generic output format is used.

defrs

This declaration is used to define a set of rules. This set of rules (rule set) can then be given a priority and also a current state. A rule set can have zero or more rules with an agenda that is local to it (local agenda). In addition, a rule set can have its own local variable declarations that must appear before any of its rules' definitions.

Syntax

```
defrs <name> {

       [ priority = <integer constant> ; ]

       [ state = active | inactive ; ]

       [ block declarations ]*

       <defrule declaration>+

       }
```

Description

A rule set's declaration section is identical to that of a compound statement's. In terms of lexical scoping, a rule set is treated as the parent block of its rules' bodies.

The **priority** declaration sets the rule set's default priority. This priority value is used to position the rule set in the primary agenda, and it can be dynamically changed at runtime (see **activate**).

The **state** declaration sets the rule set's default state. A rule set's state can either be active or inactive. An inactive rule set doesn't appear in the primary agenda, and all of the assertions sent to it are placed into its input queue (no pattern matching). When an inactive rule set becomes active, all of the assertions in its input queue are pattern matched. When an active rule set becomes inactive, it is simply removed from the primary agenda.

Fact retractions are performed regardless of a rule set's state. This retraction may be as simple as removing the fact from the input queue, or it may require a full retraction if the fact has already been pattern matched.

defrule

General Syntax

```
defrule <name> [ <rule type specifiers> ]
{

    [ summary = <string constant> ;  ]
    [ priority = <integer constant> ;      ]

    <pattern variable declarations>

      Left Hand Side (LHS)

[ => | <= ]

      Right Hand Side (RHS)

}
```

Description

The **summary** declaration is used to associate a string constant with a rule's compiled definition. At runtime this string can be accessed by the inference engine function library. Thus an inference engine function can give a verbose description of a rule, which is useful for tracing (explaining) an inference process. This value defaults to the empty string.

The **priority** declaration gives a rule a static priority value that is accessible from the inference engine function library. This value defaults to zero.

A pattern variable declaration can only be one of the following types: match, segment, or single field (integer, real, symbol, char). All pattern variables are **auto** variables.

In a forward chaining rule, the LHS represents the rule's pattern logic, and the RHS is its actions. All patterns appearing within the LHS logic default to being pattern matched to the FWM. If a pattern is to be matched to the GWM then it must have a goal specifier around it:

```
(goal <pattern> )
```

In a backward chaining rule, the LHS is a single goal pattern which is matched to the GWM, and the RHS can be a subgoal list and/or pattern logic. This pattern logic is identical to a forward chaining rule's LHS pattern logic.

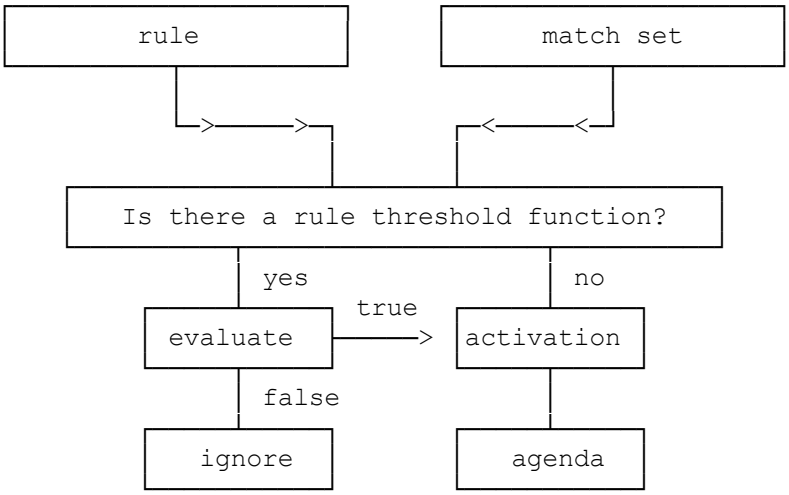
The assignment operator cannot be used within pattern logic or subgoal lists.

Overview

OPS-2000 has both forward and backward chaining rules. There is only one type of backward chaining rule, but it has three general formats. There are three types of forward chaining rules: normal form, confidence factor form, and fuzzy form. All but the normal form can utilize a threshold activation function (expression).

The LHS represents the conditions that must be satisfied in order for the rule to become **activated**. The set of data objects that cause a rule to become activated (instantiated) are referred to as the **match set**. When a rule becomes activated it is placed into a data structure called an **agenda** and is now referred to as an **activation** or **instantiation**. The set of agenda entries is referred to as a **conflict set**. A software module called the **inference engine** is responsible for operations on the agenda such as the firing and removal of activations. The inference engine selects an activation to be fired using an algorithm appropriately called the **conflict resolution algorithm**.

A threshold function (expression) is evaluated once the rule's logic has been satisfied, but before the satisfied rule is placed into the agenda as an activation. If this function has a true (nonzero) value, then its activation is placed into the agenda, otherwise the satisfied logic is ignored. This is summarized in the diagram given below.



In rule types where the system computes a value based on its pattern logic, a special variable is automatically defined to store this value, this variable is called the **logic** variable. The logic variable's name is **??**, its type is real, and its value defaults to 1.0.

Forward Chaining

```
//
//Normal Form
//
defrule <name> [ : fc ]
{
      <pattern variable declarations>

      <pattern logic>

      =>

      <compound statement>
}
```

Description

Any variable type can be declared immediately after the **=>** symbol. This is due to the fact that a forward chaining rule's RHS is a compound statement with its delimiters being the **=>** symbol and the rule's final closing brace.

The **defrule** declaration defaults to the normal forward-chaining form. When the rule's pattern logic becomes satisfied, a rule activation is placed into the agenda. There are two other forward chaining rule forms: fuzzy and confidence factor.

When a fuzzy or confidence factor rule is parsed, a special variable with the name **??** is defined for the rule's body. Its type is real, and its value defaults to 1.0. This variable is set by the system to the value of the rule's LHS pattern logic. This value is determined by the rule's definition. This variable should only be used in a rule's threshold expression and RHS. While in the scope of the rule's LHS pattern logic, the variable takes on an undefined value.

A threshold expression can optionally be specified in the fuzzy (**fz**) and confidence factor (**cf**) rule types. This is defined as a C++ expression and it can use global, rule set, expert object, and logic variables.

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 new goal WME is never compared to the existing contents of the GWM. Therefore a goal with the same data value can appear as often as it can be asserted. A goal object can have zero or more subgoal groups spawned from it. These groups are created when a backward chaining rule with an associated subgoal list is fired. A goal object can be proven by either a subgoal group being proven, or by a backward chaining rule's pattern logic being satisfied and fired.

```
defrule <name> : bc
{
      <pattern variable declarations>

      <goal pattern>

      <=

      [ <subgoal list> ]
                  - or -
      [ <pattern logic> ]
                  - or -
      [ <subgoal list> <=> <pattern logic> ]
                  - or -
      [ <pattern logic> <=> <subgoal list> ]
}
```

Description

If the goal pattern is matched (LHS), then the RHS becomes activated. Only rule variables that are bound in the LHS of a rule can be used in that rule's RHS. There are three possible types of RHSs.

Type 1: subgoal list

When a parent goal is matched, the subgoal specification list is asserted as a subgoal group, which when proven will prove the validity of the parent goal. If any goals in the group are refuted, then the entire group is refuted, and if this was the parent goal's last subgoal group, then the parent goal is refuted provided it hasn't matched any backward chaining rules with pattern logic.

A subgoal list takes the form:

```
(     ( <subgoal-1> ) ... ( <subgoal-n> )  )
```

A subgoal list must have at least one subgoal specification. A subgoal element must be of type symbol, integer, or real, and it can be specified in one of the following formats:

1. constant/literal
2. Variable bound in a rule's LHS: normal, segment, and match.
3. **(** <expression> **)**

An example:

```
//
//Backward
//
defrule Backward : bc
{
      match goal;
      real x;
      segment s;

      $goal <- (employee ?x $?s)

<=

      (     (child1 ?x)
            (next $goal : format1 end)
            (again $?s ?x $goal 10 99.9) )

} //Backward
```

Type 2: pattern logic

A backward chaining rule's pattern logic is interpreted exactly as if it were a normal-form forward-chaining rule's pattern-logic.

If the goal pattern is matched such that all of the variable names it shares with the pattern logic have the same values, then an activation is placed into the agenda such that when it fires, the corresponding goal object will be proven. Consequently, the application of the "prove" is completely dependent upon when the rule is fired. In terms of control, this is completely the opposite of how a subgoal list is applied, for in that case when a goal matches the rule's LHS, an activation is placed into the agenda, and when fired, its subgoal list will be asserted as a subgoal group which when proven or refuted will perform operations independent of the agenda. In type 1 the agenda serves as a starting point, and in type 2 the agenda serves as a completion point.

When a goal matches a goal pattern, the system gives the goal an anonymous subgoal group for the rule's pattern logic. Consequently, if a goal matches both a type (1) and type (2) rule, then the goal cannot be refuted by a subgoal group since a pattern logic subgoal group exists for the life of the goal.

Type 3: subgoal list and pattern logic

This provides a mechanism to allow for a goal to be proven using simultaneously types (1) and (2). Its behavior parallels that of two rules, one of type (1) and the other of type (2), both of which fire off the same goal pattern match. If either the pattern logic or the subgoal list are proven, then the goal is proven.

Search

OPS-2000's backward chaining doesn't constrain the developer by using a fixed backward chaining search algorithm. To understand how backward chaining works, one must first understand the basic algorithm deployed.

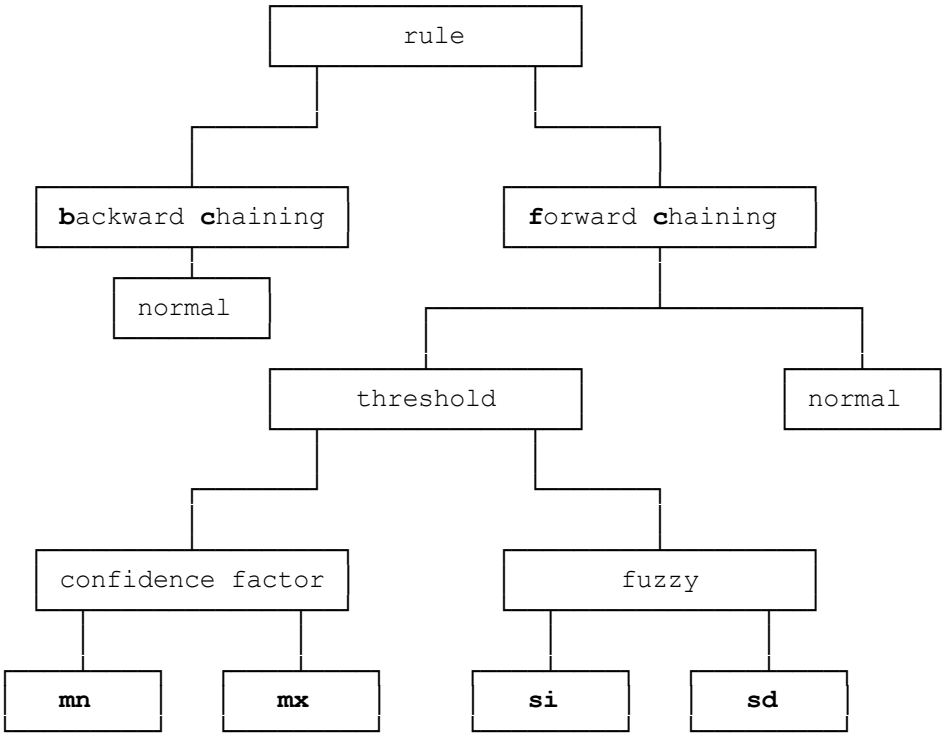
When a backward chaining rule fires, one of two things can occur:

1. If the activation was due to pattern logic, then the goal is proven.
2. If the activation was due to an associated subgoal list, then that subgoal list is atomically asserted into the GWM as a subgoal group.

Since an entire subgoal list is asserted before control is given back to the inference engine, this creates a series of breath-first activations based on the new subgoal group's members. However, since rule's are selected to be fired based on the recency of their match set, then the last subgoal specified in a subgoal list will, assuming it is a member of a match set and independent of any rule priorities, cause the next rule to be fired. If this firing asserts another subgoal group, then the cycle continues to follow a depth-first search algorithm. This of course can be changed by giving rules different priorities. For instance a rule that only has pattern logic on its RHS may be placed in a rule set with a higher priority than other rule sets which have rules with subgoal lists. This would truncate any unnecessary subgoal groups from being asserted.

Type Summary

The below diagram summarizes the rule type hierarchy.



Patterns

There are three types of patterns: free-form fact, relation fact, and class.

Free-Form

A free form pattern is used to match a WME that is neither a class object nor a relation object.

Syntax

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

Description

This pattern is satisfied once for each possible way its pattern elements can match.

Relation

A relation pattern is used to match a particular relation's working memory elements.

Syntax

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

The number of pattern elements and their associated types must match that of the relation's generic format.

Description

This pattern is satisfied once for each possible way its pattern elements can match.

Class

A class pattern is used to match a particular class's working memory elements.

Syntax

```
{ <class name> [ ^<member attribute>  <pattern element> | <class pattern> ]* }
```

Description

A class pattern can have zero or more attributes. If no attributes are specified, then the pattern matches all of the class's working memory elements. Attributes do not have to appear in any particular order.

An attribute is treated as though it were nested within an operation of the pattern's class. Thus it can access the public and private members of the pattern class exactly as though it were a class operation.

Attributes

The carat symbol **^** is used to prefix each class pattern member primary expression. The **this** pointer can optionally

be prefixed to a primary expression. Therefore in a class pattern, **^**<member> is equivalent to **^this->**<member>. If the member is of type class object, then a single '**&**' can be prefixed to the attribute. For example:

```
class usa {
      class country nation;
      };

{usa ^&this->nation {country ... } ... }
```

When a member attribute's value is of type class pointer, then a class pattern of the pointer's class type must appear immediately after the member attribute specification. This nesting of class patterns can continue indefinitely. All variable bindings apply across all nested patterns. Variable bindings occur from left to right. If at runtime a class pointer evaluates to NULL, the pattern match fails.

Pattern Element

A pattern element is used to match a data element vector. This vector can be composed of zero or more contiguous data elements. If the match is successful, then the next pattern element in the pattern is compared to the next data element vector. This continues until either a pattern element has failed to match, or the pattern and/or data object's elements have been depleted. If all of the specified data values have matched all of the pattern's elements, then the pattern has successfully matched the data object.

The pattern element can be as simple as a constant (literal match) in which case it matches a data element vector of size one. Some examples of simple free-form fact patterns are given below.

pattern	fact	matches
(Urbana is in Illinois)	(Urbana is in Illinois)	yes
(vienna is in Austria)	(Vienna is in Austria)	no
(count from 1 to 10)	(count from 1 to 10)	yes
(count from 1 to 10)	(count from 1.0 to 10)	no
(count from 1 to 10)	(count from 1 to 1.0e10)	no

Predicate

The predicate element is used to test a data element for a particular attribute. The predicate can be a check for a particular element type, or whether or not a data element is an even or odd integer.

Syntax

```
:<predicate>
```

Defined predicates are:

predicate	description
real	matches a real
symbol	matches a symbol
integer	matches an integer
number	matches either a real or integer
even	matches an even integer
odd	matches an odd integer

Examples

pattern	fact	matches?
(:number is a number)	(10 is a number)	yes
(:number is a number)	(11.0 is a number)	yes
(:number is a number)	(rocket is a number)	no
(:even)	(11)	no
(:odd is odd)	(10 is odd)	no
(:even)	(10)	yes
(:symbol)	(hello_earth)	yes
(:symbol)	(99.9)	no

Wildcard

Wildcards are used within patterns to match data elements independent of their types.

Single Field

A single field wildcard has two properties:

1. It matches a single data element.
2. It matches any type of data element.

Within patterns the symbol **?** is used to represent this wildcard. When a C++ variable is used within a rule, its name must have a **?** prefix. A variable name appearing after this wildcard serves as a type qualifier. Given below are some examples using free-form facts and patterns.

```
      pattern            fact                              matches?
=============================================================================
(count from ? to ?)      (count from 1 to 3)               yes
(count from ? to ?)      (count from 1.0 to 3)             yes
(? ? ? ?)                (count from 1 to 3)               no
(? ? ? ?)                (Lisbon is in Portugal)           yes
(? ? ? ? ?)              (1 two 3.0 four 5)                yes
```

Multiple Field

A multiple field wildcard is used to match zero or more contiguous data elements which are referred to as a segment. Within patterns the symbol **$?** is used to represent this wildcard. This wildcard can only be used in free-form patterns.

```
      pattern            fact                              matches?
=============================================================================
(the ball $?)            (the ball is red and blue)        yes
(the ball $?)            (the ball is round)               yes
(the ball $?)            (the ball)                        yes
(the ball $?)            (the)                             no
($? $?)                  (the)                             yes, yes
```

The last pattern matches twice, for each wildcard takes its turn matching the data element. An efficient use of this wildcard is to use it as a pattern's last element. However when embedded elsewhere in a rule, it should be used with prudence.

This wildcard cannot be used with pattern elements that match a single data element.

Variables

Variables are bound in a rule in descending order. Consequently, a rule's first pattern inherits no variable bindings.

Rule variables can be any legal C++ variable, and their declarations are identical to those of C++. However within the scope of a rule, all OPS-2000 words are of type symbol. Therefore all C++ variable names used within a rule must be prefixed with a '**?**'. The '**?**' is also the single field wildcard operator.

For example:

```
/*
 *hello
 */
defrule hello
{
      integer a;         //C++ variable declaration.

      (?a is an integer)

=>

      ?a += 10;

      printf("This value of ?a is %d.\n", ?a);
      printf("This symbol value = %s.\n", hello);

} /*hello*/
```

The table given below gives some examples of pattern matches using this type of variable.

```
integer a, b;
symbol x, y;

      pattern                  data object
================================================================
(?x is the brother of ?y)      (Dexter is the brother of Eugene)
                                          ?x = \Dexter\
                                          ?y = \Eugene\

(There are ?a dollars)         (There are 1000 dollars)
                                          ?a = 1000

(arc ?a to ?b)                 (arc 11 to 1000)
                                          ?a = 11
                                          ?b = 1000

($? ?a $?)                     (primes are 11 17 23)
                                          ?a = 11          //Match one
                                          ?a = 17          //Match two
                                          ?a = 23          //Match three
```

Segment

A segment variable is used to store a segment matched by the multiple field wildcard operator.

Segment variable rules:

1. In use, a segment variable name must be prefixed with the multiple field wildcard symbol **$?**.
2. A segment variable will match zero or more fact elements. Thus it represents the bound form of the multiple field wildcard operator.
3. A segment variable cannot appear within a disjunctive term.
4. A segment variable can be used in three places: free-form patterns, **assert** statements, and **printout** statements.

Match

A match variable is used to store a data object that has matched a particular pattern condition. In use, variables of this type are prefixed with the **$** symbol.

There are two types of match variables: class and fact. The compiler, based on a variable's actual usage, binds one of these types to the match variable's definition.

A match variable can only be bound to one match type throughout a rule's definition. This is due to the fact that a match variable's type is created based on the type of the first pattern it is bound to within the text of a rule's definition.

A class match variable also has a particular C++ class it is bound to. Consequently, these variables can be treated as though they were actually declared as pointers to their binding C++ class, which means that the member and pointer operators can be used on them.

Any type of match variable can be used in **assert**, **refute**, **retract**, **reassert**, and **printout** statements. This is in addition to any other defined uses.

Match variables can be used in test conditions, but not in pattern conditions.

Operators

There are three types of operators that can be used within a pattern: **~** (not), **&** (and), and **|** (or). The **~** has highest precedence, followed next by the **&**, and then the **|**.

Negation

The negation operator is represented by the symbol **~**, which means take the complement of the pattern element match.

Syntax

```
~ <pattern element>
```

Description

A negation indicates to the system that if the element matches, then it fails, and likewise if it fails to match, then it succeeds (complement of the normal pattern element match).

Within a pattern, a variable name cannot be negated more than once before it is bound in that same pattern (please note). For example:

```
(~?x ?x ~?x)       //legal
(~?x)              //legal
(~?x ~?x)          //illegal
```

```
(~?x ~?x ?x)       //illegal
(?x ~?x ~?x)       //legal
(?x ~?x ?x ~?x)    //legal
(~?x ~?x | ?x)     //illegal
(~?x | ~?x ?x)     //illegal
(~?x | ?x ~?x)     //legal
```

The last pattern is legal, but it is actually treated as though it were the pattern: (?x ~?x). This is because at runtime if a variable has two negations before it has a successful binding, the pattern matching for that particular instance fails. The last pattern can create two possible pattern matching instances, however one of them is guaranteed to always fail.

If a variable is negated before it is bound in a pattern, then it must be bound in a previous pattern. For example:

```
(... ?x ...)
(~?x)              //legal only if the above condition exists.
```

Disjunctive

The disjunctive operator is represented by the symbol **|** (pipe), which means each of its operands creates one possible match for a particular data element.

Syntax

```
<pattern element> [ | <pattern element> ]*
```

Description

All variables appearing within the disjunction must have been previously bound. A variable cannot be bound as an element of a disjunctive term.

Segment variables cannot appear in disjunctive terms.

Order of evaluation is from left to right. All operands are always evaluated.

Example:

```
(?x | ?y | ?z)
```

The above example will match a single fact element with the value of ?x or ?y or ?z. If all three variables have the same value, then a matching fact element will match three times (please note).

Conjunctive

The conjunctive operator is represented by the symbol **&** (ampersand), which means everything in the conjunction must match the particular data element vector.

Syntax

```
<pattern element> [ & <pattern element> ]*
```

Description

Order of evaluation is from left to right. Evaluation ceases when either the clause has ended, or a pattern element doesn't match.

It is illegal to use rule variables of different fundamental types in the same conjunctive term.

It is illegal to use **:**<pred> with the multiple field wildcard.

Any unbound variables appearing in the conjunction are bound to the matching data element vector.

Example:

```
(?x ?y ?x&?y)
```

The above example will match a three element fact object such that all three elements are identical. For example:

```
(hello hello hello)            //Matches
(hello goodbye hello)          //Doesn't match
```

Test

Intrapattern test.

Syntax

:(<test expr> **)**

The test expression should have a result of type integer. Note that relation and logical operators have integer results.

Description

If the test expression evaluates to true (noninteger zero) then the term matches, otherwise it fails.

A test expression indicates to the OPS-2000 compiler that at a specific point in the matching of a data object to a pattern, the test should evaluate to true. Consequently only the bindings occurring to the left of the test expression, within the pattern, can be used within it. A test must appear in a conjunctive term that contains some nontest term. This term can be a predicate.

Interpattern testing should be done using the pattern logic **test** condition.

Example:

```
(?x ?y&:((?y == hello) || (?y == good_bye)))
```

If the test term appears within a class pattern, then the expression can directly access the members of the matching class object via the **this** pointer.

```
{employee ^name ?name&:(this->process( ?name ) )  }
```

```
(?x & :(?x > 1))         //Example of embedded test.
```

Priority

The ordering of an activation in a rule set's local agenda is based on two criterion:

1. The rule's declared priority.
2. The match set with the most recent WME.

A rule's priority can be declared by placing a **priority** declaration in the rule's primary variable declaration section. This section appears immediately after a rule's first opening brace.

```
priority = <integer constant>;
```

For example:

```
/*
 *PriorityExample
 */
defrule PriorityExample
{
    priority = 20;
    int a, b, c;

    (?a ?b ?c)

=>

      printf("%d %d %d\n", ?a, ?b, ?c);

      return;

} /*PriorityExample*/
```

Logical Operators

There are three types of logical pattern operators: **and**, **or**, and **not**. These operators have operands that appear using a postfix notation. The operands are referred to as slot specifications which can be any one of the following: a pattern condition, a test condition, or **any** logical pattern operator.

General Syntax

```
( <operator> <operand>+ )
```

And

A logical **and** is satisfied only if each and every one its slot specifications are satisfied.

Syntax

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

Description

A logical **and** is satisfied only if each and every one its slot specifications are satisfied.

A rule's pattern logic always has an implicit logical **and** enclosing it. For example:

User Input Form	OPS-2000 Internal Form+

/*
* America
*/
defrule America
{
 	
(initial_fact)
(a land of dreamers)
(count down 3 2 1)
 	
=>
 	
assert(Dreamers);
 	
} /*America*/
	/*
	* America
	*/
	defrule America
	{
	
	(and (initial_fact)
	(a land of dreamers)
	(count down 3 2 1))
	
	=>
	
	assert(Dreamers);
	
	} /*America*/

Or

The logical **or** is used to specify multiple slot specifications that can make a single slot specification satisfied.

Syntax

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

Description

The logical **or** is satisfied each time one of its slot specifications is satisfied (please note). This differs drastically from the traditional procedural language's logical **or** where if any condition is true then the logical **or** is satisfied exactly once. Here, if every slot specification is true, then the **or** is satisfied once for each of them.

Test

The rule test condition is used to test for values in a rule's pattern logic. This is a interpattern test.

Syntax

```
(test ( <expression> ) )
```

Description

If the expression evaluates to a nonzero value then the test condition is true.

If at compile time a test is proceeded by a logical **not**, then the logical **not** is replaced by the C++ unary not. For example:

```
(not (test (?x > ?y)))

is replaced by

(test ( !(?x > ?y)) )
```

Not

The logical not is used to check that a particular condition does not exist. If this condition is a pattern condition, then the check is for the nonexistence of a particular data object pattern. If this condition is a test condition, then the check is for the falsity of the test condition's expression.

Syntax

```
(not <slot specification>)
```

Logical **not** can appear anywhere in a pattern logic expression and it must have exactly one operand.

Description

At runtime the logical not is applied to test and pattern conditions. Therefore at compile time DeMorgan's Theorem is applied to all pattern logic to simplify the logic so that only test and pattern conditions have a **not** appearing before them.

A **not** pattern is satisfied if and only if no working memory elements exactly match the pattern. If a **not** has variables, then each set of variables provides a way for the pattern to be satisfied. If given a variable set, the pattern doesn't match any working memory elements, then the **not** is satisfied for this particular variable set.

All variables appearing within a **not** pattern must have been previously bound in another pattern condition.

DeMorgan's Theorem

A summary of this theorem is given below in terms of slot specifications.

```
(not (not <slot> ))
is
<slot>


(not (and <slot1> <slot2>))
is
(or (not <slot1>) (not <slot2>))


(not (or <slot1> <slot2>))
is
(and (not <slot1>) (not <slot2>))
```

Expressions

An expression consists of one or more operands with an operator. Note that an expression followed by a semicolon is a statement.

Operand Notation:

```
e     - any expression
v     - any expression a value can be assigned to.

prefixes:

      i     - integer (int or char).
      a     - arithmetic (int, char, double)
      p     - pointer
      c     - class
      cp    - class pointer
```

Operators

Assignment Operators

Operator	Usage	Description
=	v = e	Assign the value of e to v.
+=	av += ae	av = av + ae
-=	av -= ae	av = av - ae
*=	av *= ae	av = av * ae
/=	av /= ae	av = av / ae
%=	iv %= ie	iv = iv % ie
<<=	iv <<= ie	iv = iv << ie
>>=	iv >>= ie	iv = iv >> ie
&=	iv &= ie	iv = iv & ie
|=	iv |= ie	iv = iv | ie
^=	iv ^= ie	iv = iv ^ ie

The value of v and e is only fetched once. For example in your code "v += e" is more efficient then "v = v + e", for in the latter the value of v is determined twice, while in the former it is only done once.

Arithmetic Operators

Operator	Usage	Description
+	ae1 + ae2	Sum of ae1 and ae2
-	ae1 - ae2	Difference of ae1 and ae2
*	ae1 * ae2	Product of ae1 and ae2
/	ae1 / ae2	Quotient of ae1 and ae2
%	ie1 % ie2	Remainder of ie1 / ie2
+	+ ae1	Positive ae1
-	- ae1	Minus ae1
++	++ iv1 iv1 ++	Increment by 1 Take value and then increment by 1
--	-- iv1 iv1 --	Decrement by 1 Take value and then decrement by 1

Bit Operators

Operator	Usage	Description
>>	ie1 >> ie2	ie1 shifted right by ie2 bits
<<	ie1 << ie2	ie1 shifted left by ie2 bits
&	ie1 & ie2	Bitwise "and" of ie1 and ie2
|	ie1 | ie2	Bitwise "or" of ie1 and ie2
^	ie1 ^ ie2	Bitwise exclusive "or" of ie1 and ie2
~	~ie	One's complement of ie

Relational Operators

Operator	Usage	Description
<	ae1 < ae2	True if ae1 is less than ae2
<=	ae1 <= ae2	True if ae1 is less than or equal ae2
>	ae1 > ae2	True if ae1 is greater than ae2
>=	ae1 >= ae2	True if ae1 is greater than or equal ae2

Equality Operators

Operator	Usage	Description
==	ae1 == ae2	True if ae1 is equal to ae2
	pe1 == pe2	True if pe1 is equal to pe2
!=	ae1 != ae2	True if ae1 is not equal to ae2
	pe1 != pe2	True if pe1 is not equal to pe2

Logical Operators

Operator	Usage	Description
&&	e1 && e2	True if e1 and e2 are nonzero. Evaluation is from left to right, and the first zero valued expression stops the evaluation loop.
||	e1 || e2	True if e1 or e2 are nonzero. Evaluation is from left to right, and the first nonzero valued expression stops the evaluation loop.
!	!ae	True if ae is false.

Conditional Operator

Operator	Usage	Description
?:	ae ? e1 : e2	If ae or pe is true, e1 is evaluated.
	pe ? e1 : e2	If ae or pe is false, e2 is evaluated.
		e1 and e2 must both be of the same type.

Comma Operator

Operator	Usage	Description
,	e1 , e2	Sequentially evaluates e1 and then e2. The value of the expression is that of e2's. The comma is also used as a delimiter. To use a comma operator in a function call parameter, the parameter expression must be enclosed in parenthesis. This also applies to any other features of the language that use a comma as a delimiter.

Examples

```
function call:     foo(1 + 1, (2 + 19, 91));
assert statement:  assert((1 + 1, 2 + 2), "OPS");
```

Address Operators

Operator	Usage	Description
&	&ve	Address of an expression that refers to an object.
*	* pe	Contents of address pe.
[]	pe[ie]	A variable offset ie variables from the address given by pe.

Class Operators

Operator	Usage	Description
.	cv . cmem	Member cmem of class cv.
->	cpe -> cmem	Member cmem of class pointed to by cpe.

Precedence and Associativity

Unary operators and assignment operators are right associative; all others are left associative. For example:

```
a = b = c;          means        a = (b = c);

a + b + c;          means        (a + b) + c;
```

The table given below summarizes precedence. Operator precedence decreases as the operator appears towards the bottom of the table. The operator at the top of the table has the highest precedence. Operators with equal precedence appear grouped in the same table row.

Operator	Description
::	global qualifier
->	member selection
[]	subscripting
()	function call
~	complement
!	unary not
++	pre-increment
++	post-increment
--	pre-decrement
--	post-decrement
-	unary minus
+	unary plus
&	address of
*	dereference
new	create (allocate)
delete	destroy (de-allocate)
delete []	destroy vector
*	multiply
/	divide
%	modulo (remainder)
+	add
-	subtract
<<	shift left
>>	shift right

Operator	Description
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
!=	not equal
&	bitwise AND
^	bitwise exclusive OR (XOR)
|	bitwise inclusive OR
&&	logical AND
||	logical inclusive OR
? :	conditional expression
=	simple assignment
+=	add followed by assignment
-=	subtract followed by assignment
*=	multiply followed by assignment
/=	divide followed by assignment
%=	modulo followed by assignment
<<=	shift left followed by assignment
>>=	shift right followed by assignment
&=	bitwise AND followed by assignment
|=	bitwise OR followed by assignment
^=	bitwise XOR followed by assignment
,	comma expression

New

The **new** free store operator is used to create dynamic objects. The lifetime of an object created by **new** is not restricted to the scope in which it is created. The **new** operator returns a pointer to the object it created. A constructor is always called on every class object created.

Syntax

This operator has three general forms.

Form 1:

```
new ( <type name> )
```

This form is used to create a single object of a particular type. If that object happens to be a class object, then the respective class's no parameter constructor is used to initialize the object.

Examples:

```
new (integer)
new (class date *)
new (class date)
```

Form 2:

The type specification is similar to that of form-1, except now zero or more vector dimensions can be specified.

```
new <type name> [ [ <integer expression> ] ]*
```

The integer expression is used to specify a dimension of an object. At runtime each expression is evaluated, and if the values are less than one, the operation aborts. The **new** term returns an empty (null) pointer when a runtime error occurs.

Examples:

```
new class date[100]           //Vector of 100 class objects.
new integer[200][200]         //200 x 200 array of integers.
new integer **[200]           //Vector of 200 (integer **).
new integer                   //Single integer.
```

Form 3:

This form is used to create a single class object. The **new** operator uses the specified constructor to initialize the new class object.

```
new <constructor>
```

Examples:

```
new date()
new date(1, 1, 2000)
new date(4, 19, 1964)
new date("January 1, 2000")
```

Delete

The **delete** operator destroys an object created by the **new** operator. **Delete**'s operand **must** be a pointer returned by the **new** operator (please note).

Syntax

```
delete [ [ <expr1> ] ] <expr2>
```

If expr2 is of type (class *), then expr1 can optionally be used to specify the length of the vector to be deleted. This is done so that a class's destructor can be applied to every element of an arbitrarily sized vector.

Description

Delete can only destroy a single dimension of a data structure. This vector, provided no exceptions occur, is always completely freed. Never use **delete** on an address that was not directly returned by **new**.

To free a multi-dimensional array allocated by **new**; **delete** must be called once for each dimension of the array.

OPS-2000 will generate runtime errors if either a size expression's value exceeds the size of the vector or if the value is negative. If the value is zero, then delete does not free the vector. If no size expression is given, and the vector is a class object, then the system assumes it is of length one and applies the appropriate destructor.

The delete term first evaluates any size expression, and then checks the properties of the vector. If the vector is NULL, then delete does nothing. Otherwise it continues by checking for one of the possible runtime errors described above. Next if applicable it applies a destructor. Lastly it frees the vector.

Deleting an object that has already been deleted will corrupt the system. Thus immediately after a vector has been deleted, all references to it should be removed.

Functions

In the current version of OPS-2000, all function parameters are passed by value. The C++ language has a pass by reference option which is not supported in the current OPS-2000 release.

A function name may be duplicated (overloaded), so its prototype is used to resolve the ambiguity.

A function's prototype can be given before its actual definition is specified. This prototype indicates the function's parameter types and return type specifications. A function prototype cannot have named parameters, its parameter specification consists solely of each parameter's type. For example:

```
real Sort(integer, integer *, char *);
```

There are two basic types of functions: class and general.

A class function definition is local to its class's definition and can either be an operation, constructor, or destructor. These are declared by specifying the function in the class definition, which can also include a function definition. If a class function's definition is given outside of the class specification, then the function specification name must be prefixed with a class name qualifier which takes the form:

```
 <class name> :: <function name>.
```

An example:

```
//
//employee :: Print()
//
integer employee :: Print ()
{
    return( 1 );
}
```

Globals

A global variable can be accessed directly from a function body definition by using the global qualifier. This resolves any potential scope problems.

```
::<variable usage>
```

Inference Engine Function

The inference engine function is used to control the inferencing process. This function uses a set of system defined functions to perform actions such as the firing and removal of activations. There is a wealth of functions that provide information on everything from an expert object's current state to an activation's match set size.

Library

Given below is a summary of the inference engine function control library. Formal descriptions of these functions is given at the end of this manual.

EO - Expert Object

PA - Primary Agenda

LA - Local Agenda.

IE - Inference Engine

RS - Rule Set

CS - Conflict Set

MS - Match Set

WM - Working Memory

WME - Working Memory Element

ACT - Activation

Function	Description
ie_eo_stats	Provides general IE statistics.
ie_watch	IE watch flag value.
ie_eo_wm_cmp	Sets EO's flag for comparing WME's.
ie_eo_wm_size	Size of a particular working memory.
ie_eo_wme_remove	Retract a WME.
ie_eo_stop	Stops the current EO.
ie_eo_is_stopped	Is the current EO stopped?
ie_pa_rs_size	Size of the EO's PA.
ie_pa_rs_info	Information about a PA rule set.
ie_pa_rs_index_by_sys_id	PA index for the RS id.
ie_pa_rs_index_by_name	PA index for the RS name.
ie_cs_size	A RS's conflict set size.
ie_cs_fire	Fire an activation.
ie_cs_remove	Remove an activation.
ie_cs_fire_remove	Atomically fire & remove an activation.
ie_act_rule_info	General information about the rule.
ie_act_rule_stats	Information about the activated rule.
ie_act_nth_ms_id	Returns WME id for the nth MS member.
ie_default	Specify the default inference engine.

Statement

The elementary actions of evaluation, assignment, and control of evaluation order are specified by the statements of a programming language. OPS-2000 has two groups of statements: those that can appear anywhere a statement is permitted (for, while, do-while, if-else, return, continue, break, switch, case, default, expression, compound, empty), and those that can only appear within an expert object (printout, excise, stop, assert, retract, reassert, refute, send, receive, activate, deactivate).

Activate

The activate statement is used to activate a rule set. It can also be used to dynamically change a rule set's runtime priority. This statement can only appear within an EO, and its actions can only affect a rule set within that EO.

Syntax

```
activate <symbol expr> [: <integer expr> ] ;
```

Description

The symbol expression is evaluated for a rule set's name. If the rule set name exists and the corresponding rule set is inactive, then its state is set to be active, and any data in its input queue is pattern matched to its rules. If the rule set name doesn't exist, then a runtime error is flagged.

An activate statement can also change a currently active rule set's runtime priority (dynamic runtime priorities). The integer expression's value is the active rule set's new priority value.

Assert

The assert statement is used to assert a data object into one of three destinations: FWM, GWM, or a character buffer.

Syntax

This statement has two general forms.

```
Form 1:     assert( <element> [, <element> ]* ) [ : <real expr> ]
                  [ => [ <symbol expr> | <char pointer expr> ] ];

Form 2:     assert <class pointer expr> [ : <real expr> ] ;
```

Description

Form 1:

This form is used to assert facts into one of three destinations: **FWM**, **GWM**, or a character buffer.

An assert element can be one of the following: match or segment variable, integer, real, symbol, or pointer to character.

If an element's final value is a match object, then that match variable has to be bound to a fact object. The match variable can have an optional relation output format specifier proceeding it: $<variable> **:** <format>. If at runtime the match variable is bound to a relation object, then that object's relation definition is used for the search for the specified output format. If the format exists, then it is used, otherwise the relation's generic output format is used.

The symbol expression must take the value of either **FWM** or **GWM**.

If a fact is asserted into a buffer, then the buffer will take the value of the fact's string representation.

Form 2:

This form is used to assert a class object into the FWM. If the evaluated expression is NULL, then nothing is asserted. The certainty expression is always evaluated.

Break

Syntax

```
break;
```

Description

The break statement causes the termination of the smallest enclosing **switch**, **while**, **do**, or **for**. Control passes to the statement immediately following the terminated statement.

Compound

A compound statement is used to group declarations and statements. This grouping is often referred to as a block.

Syntax

```
{
        [ <declaration> ]*
        [ <statement> ]*
}
```

This syntax allows for blocks to be nested.

Description

A block's declarations are evaluated each time it is entered, meaning the block's objects are created and initialized each time it is entered.

All block references are bound at compile time (static binding). This is also referred to as lexical scoping.

The conventional rules of lexical scoping are:

1. The scope of a declaration includes the block in which it occurs but excludes any surrounding block.
2. The scope of a declaration includes any block contained within the block in which the declaration occurs, but excludes any contained block in which the same variable name is redeclared.

A function's body and a rule's body are two instances of where a block specification has been used in the OPS-2000 syntax.

Continue

Syntax

```
continue;
```

Description

The **continue** statement causes control to pass to the loop continuation portion of the smallest enclosing **while**, **do**, or **for**.

Deactivate

The deactivate statement is used to transfer a rule set to an inactive state. Any new assertions are placed into the rule set's input queue. The rule set still processes retracts which can be as simple as removing a data object from the rule set's input queue, or as complicated as removing the data object from the rule set's pattern matching network. An inactive rule set will not appear in the primary agenda.

This statement can only appear within an EO, and its affect is limited to its enclosing EO's rule sets.

Syntax

```
deactivate <symbol expr> ;
```

Description

The symbol expression is evaluated for a rule set name. If this rule set exists then its state is set to inactive, otherwise a runtime error is flagged.

Do-While

Syntax

```
do <statement> while ( <expr> );
```

Description

The flow of control is: first the statement is executed, then the expression is evaluated. If the expression's value is true (nonzero), then the loop is continued, otherwise it is exited.

Excise

Syntax

```
excise <symbol expr> ;
```

Description

At runtime the symbol expression is evaluated. If the rule name exists in the firing rule's rule set, then the rule is removed from the system, otherwise the system ignores the request.

Expression

Syntax

```
<expression> ;
```

Description

The expression is executed.

For

The **for** statement is used for loop iteration.

Syntax

```
for ( <statement1> [ <expr1> ] ; [ <expr2> ] )
      <statement2>
```

Description

Statement1 serves as the loop's initializer. Expr1 is evaluated before each execution of the loop's body, if its value is false (zero) then the loop is exited. Statement2 is the loop's body. Lastly, expr2 is always executed immediately after statement2's execution.

If-Else

The conditional statement is used for conditional statement execution.

Syntax

```
if ( <expr> )
      <statement1>
[ else
      <statement2> ]
```

Description

If the expression is true, then statement1 is executed, otherwise if statement2 is specified then it is executed.

As usual the "else" ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

Null

Provides a mechanism to specify an empty statement (no actions).

Syntax

```
;
```

Description

This statement has no meaning.

Examples

```
integer i;

//A simple loop to consume CPU time.
for (i = 0; i < 2000; i++);

//An infinite loop
for (;;)
    printf("Hello World\n");
```

Printout

Prints out a line of text followed by a carriage return.

Syntax

```
printout ( [ <element> [, <element> ]* ] );
```

Description

An element can be one of the following: match or segment variable, integer, real, symbol, or pointer to character. If the element list is empty, then the printout acts like a carriage return.

If an element's final type is match object, then that match variable has to be bound to a fact object. The match variable can have an optional relation output format specifier proceeding it: $<variable> **:** <format>. At runtime, if the match variable is bound to a relation object, then that relation's definition is searched for the output format name. If the format exists, then it is used, otherwise the generic output format is used.

Reassert

Efficiently reperforms a FWM element's pattern matching. Goals cannot be reasserted.

Syntax

```
reassert <match variable> [ : <integer expr> ]
             [, <match variable> [ : <integer expr> ] ]* ;
```

Description

The match variable must have been bound in the rule's pattern logic, otherwise a runtime error is flagged. The bound object is retracted and then asserted. The original object is not destroyed.

The integer expression is so that a reasserted data object can be given a new certainty, if this value is not specified then the new assertion will have the same certainty as the original.

If the match variable was bound to a goal, then it is simply ignored. This is not a runtime error.

Receive

The receive statement is used to receive messages from a channel.

Syntax

```
receive( <channel name>, <char pointer expr>, <integer expr> );
```

The channel name is evaluated at compile time and it must be a symbol constant.

Description

The channel name must be declared before it is used. Only one EO can receive from a particular channel, but there is no actual limit on the number of receive statements an EO can have for a particular channel.

If there is an input message then it is placed into the character buffer specified by the character pointer expression. If there is no input message, then the sending EO is executed (**run()**) with the integer expression's value as the steps parameter and the buffer is rechecked for an input message. If there are still no messages then byte zero of the input buffer is set to the null character: '\0'. A busy wait loop can be implemented if receiving a message is mandatory. The integer expression is only evaluated if there were no original messages in the channel.

Refute

The refute statement is used to refute a goal. Refute means to prove a statement or argument to be false or erroneous.

Syntax

```
refute <match variable> [, <match variable> ]* ;
```

Description

Goals:

If the match variable is bound to a goal, then that goal is retracted. The following rules apply:

1. If that goal was a member of a subgoal group then the entire subgoal group is retracted.
2. If (1) was the last subgoal group that the parent goal has, then the parent goal is refuted.
3. If the goal was not a member of a subgoal group, then the goal is simply removed from the GWM.

Facts:

If the match variable is bound to a fact, then that fact is simply retracted as if the **refute** was actually a **retract**. However don't confuse the two when its comes to goals, for they have completely different results. The use of **retract** on a goal object indicates that the goal has been proven.

Retract

The retract statement is used to retract a data object from its working memory. This statement can only appear within a rule's body.

Syntax

```
retract <match variable> [, <match variable>]* ;
```

Description

Facts:

A retracted fact is simply removed from the FWM.

Goals:

A retracted goal is removed from the GWM. This action indicates that the goal has been proven, if this is not the case then the **refute** statement should be used. The following rules apply:

1. If the goal has no parent, then the goal is asserted into the FWM, for it has been proven.
2. If the goal was the last member of a subgoal group then **retract** is applied to the parent goal.
3. If the goal was not the last member of a subgoal group then the goal is simply retracted.

Overview

The match variable must have been bound in the rule's pattern logic. If retract is given an unbound match variable, then a runtime error is flagged.

If the match variable is bound to a class object, then the actual class object is not destroyed. This object must be explicitly destroyed using **delete**. For example:

```
defrule Example
{
      match employee;

      $employee <- {employee ^name ? }
```

```
=>

      delete $employee;        //Delete's the class object.
      retract $employee;       //Delete's employee's WME.

} //Example
```

The use of **delete** before **retract** is not mandatory. A retracted WME's value physically exists until the rule's firing completes.

One way to determine whether or not a class object is in working memory is to have a flag in the class's definition that is explicitly set to true when the object is asserted, and later set to false when it is retracted. For example:

```
/*
 * AssertEmployee
 */
defrule AssertEmployee
{
      ($?)

=>

      class employee *object;

      ?object = new employee;
      ?object->assert = 1;
      assert ?object;

} /*AssertEmployee*/

/*
 * RetractEmployee
 */
defrule RetractEmployee
{
      match object;

      $object <- {employee}

=>

      $object->assert = 0;
      retract $object;

} /*RetractEmployee*/
```

Return

Return from a function or rule body.

Syntax

```
      return [ <expr> ] ;
```

Description

The return statement exits from the enclosing function or rule body. Its expression's value is relevant only when the statement appears in a function, for there it is treated as the function's return value. Thus a return without an expression indicates that the function doesn't return a value, and therefore it is acting as a procedure. A rule's body is modelled after this type of procedure function.

Send

The send statement is used to place a message into a channel.

Syntax

```
send( <channel name> , <char pointer expr> );
```

The channel name is evaluated at compile time and it must be a symbol constant.

Description

A channel name must be declared before it can be used. There can only be one EO sending into a particular channel. However there is no actual limit on the number of **send** statements an EO can have for a particular channel.

Stop

The **stop** statement is used to set an EO's stop flag.

Syntax

```
stop;
```

Description

This statement sets the EO's stop flag to true, and then acts like a **return** statement.

Switch

The switch statement causes control to be transferred to one of several statements depending on the value of an expression.

Syntax

```
switch ( <integer expression> )
      <statement>
```

A switch statement can have zero or more values specified using the **case** label syntax, no two of which can have the same integer constant value. If a value is matched, the statement proceeding it is branched to.

```
case <integer constant value> : <statement>
```

A switch statement can optionally have a **default** label. This label is branched to if none of the **case** values match the switch expression's value.

```
default : <statement>
```

Description

OPS-2000 has a limited form of the **switch** statement. This form is limited in that if the switch's statement is a compound statement, then its case statements cannot appear within any nested compound statements. The case constant expressions must be of type **int**, which in OPS-2000 means it can be either **char** or **integer**.

A switch statement can be seen as a special statement form such that based on the integer expression's value, a branch is made into the statement. If this statement is a compound statement, then a branch is made to a point in that compound statement. Otherwise, the branch can only be made to the start of the statement. For example:

```
switch ( x )             //noncompound statement
      case 10 :
      case 20 :
            for (i = 0; i < 10; i = i + 1);

switch ( x ) {           //compound statement
      case 10 :
            i = 30 + x;
            break;

      case 20 :
            hello();

      } /*switch*/
```

The **switch** statement recognizes the **break** statement.

While

The while statement is used for loop iteration.

Syntax

```
while ( <expr> )
    <statement>
```

Description

The expression is evaluated, if true (nonzero) then the statement is executed. This cycle continues until the expression evaluates to false.

Typedef

This declaration is used for creating new data type names. It creates a synonym for an existing type, rather than a new type. For example:

```
typedef integer INDEX;
```

The above **typedef** makes the name **INDEX** a synonym for **integer**. The "type" **LENGTH** can be used anywhere "type" **integer** can be used.

The OPS-2000 version of the **typedef** facility is limited in that it only permits a new data type name to take on a type definition that precedes the name in the **typedef**. For example:

```
typedef integer INDEX, *INDEX_PTR;   //(int), (int *)
typedef char *letter_ptr, letter;    //(char *), (char)
```

Variable

A variable declaration is a type specification followed by a variable name. For example:

```
integer a;
integer a, b, c;
char *a, **b, c[10];
char a[20];
real a[10][20][30];
real *a[10][20];
```

OPS-2000 System Functions

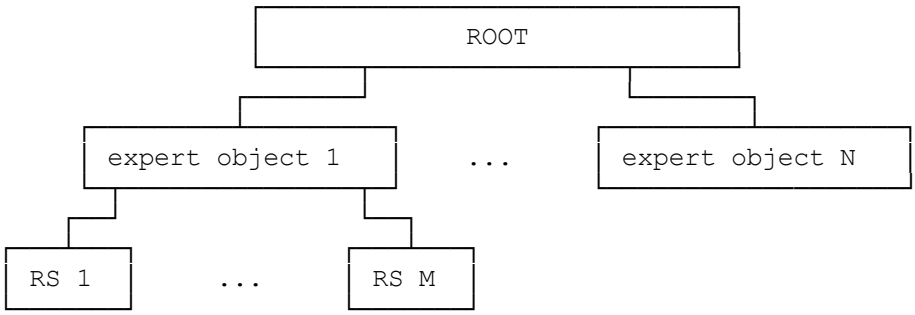
System Functions

A compiled function can return either a real, integer, char, or symbol value. Pointer values cannot be explicitly returned from a **compiled** function. Single dimensional arrays of these four types can be passed to compiled functions. Symbols returned by compiled functions are entered into the system's symbol table. A symbol returning compiled function actually returns a character string which is entered into the system's symbol table. Symbols are almost identical to string constants, except the equality operators can be applied to them and they should never be destroyed.

Operating Environment

The operating environment library is composed of functions that serve primarily as the command-line interface. Functions in this library can only be called from the interpreter environment (please note).

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. This tree-like 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 operating environments. This structure can be traversed using the **ce()** function.



Variable declarations that are local to each environment can be displayed using the **list()** function. This operating environment hierarchy allows the developer to simultaneously debug and test multiple expert objects.

Name

agenda() - display agenda

Synopsis

integer agenda()

integer agenda(path)

char *path;

Description

If **agenda()** is given no parameters then the agenda of the current expert object or rule set is displayed.

If **agenda()** is given a parameter then

1. If there is a current expert object then

   i) If parameter's value is **"primary"** then the Expert Object's primary agenda is displayed.

   ii) If not (i) then if the parameter's value is a rule set name in the current expert object, then the rule set's local agenda is displayed.

2. Otherwise the parameter is interpreted as the name of an expert object.

Name

cwe(), ce() - current environment information

Synopsis

integer cwe()

integer ce()

integer ce(cwe)

char *cwe;

Description

Function **cwe()** displays the Current Working Environment (CWE).

Function **ce()** sets the root environment to be the CWE.

Function **ce(path)** is used to change the CWE. Its parameter format is:

```
<string1> [ : <string2> ]
```

If only string1 is specified then:

1. If there is a current expert object and string1 is the name of one of its rule sets, then set the CWE to be that of the rule set's.
2. If there is a current expert object and string1 is not the name of one of its rule sets, but it is the name of another expert object, then set the CWE to be that of the expert object's.
3. If there is not a current expert object and string1 is the name of an expert object, then set the CWE to that of the expert object's.

If both string1 and string2 are specified, then string1 is interpreted as an expert object's name, and string2 is interpreted as a rule set within that expert object. If both the expert object and its rule set exist, then the environment is changed to that of the corresponding rule set's.

Name

channel() - channel information

Synopsis

integer channel(channel)

char *channel;

Description

Function **channel()** is used to display information about a particular channel. This includes its name, sender expert object, receiver expert object, and the current state of its message buffer.

Name

clear() - clear all definitions from the system

Synopsis

integer clear()

integer clear(eo_name)

char *eo_name;

Description

If **clear()** is given no parameters then the system is cleared of all definitions.

If **clear()** is given a parameter then the parameter must be the name of a noncurrent expert object. If the expert object exists, it is cleared from the system.

Name

declare() - declare a string definition

Synopsis

integer declare(definition)

char *definition;

Description

Function **declare()** is used to declare a top-level definition into the system. Remember that string constants have a limited length, while dynamic strings can be of any length.

Example

--> declare("integer a, b, c;");

--> declare("overload Hello;");

Name

edit() - interface function for editing

Synopsis

integer edit(param)

char *param;

integer edit(param)

symbol param;

Description

Function **edit()** can be used to access the editor specified by the environment variable "**editor**". The value of this variable should correspond to an executable system editor. See **env()**.

Name

env(), set() - functions for accessing and setting OE variables

Synopsis

integer env()

integer set(set_string)

char *set_string;

Description

Function **env()** is used to display the OE's variables' values.

Function **set()** is used to set an OE variable's value. Its format is:

<name>=<value>

The table given below gives a summary of the OE's variables.

Name	Description
editor	Editor name used by **edit()**.
prompt	OE prompt value.
cwe	If set to **true**, then the CWE is displayed to the left of the prompt string. When set to **false**, only the prompt string will appear.
rule	If set to **true**, then the OE statement interpreter will accept rule specific statements such as **assert**. If set to **false**, then the OE statement interpreter only accepts C++ statements.

Examples

```
-->
--> set("editor=emacs");
--> set("rule=true");
-->
```

Name

exit() - exit function

Synopsis

integer exit()

Description

Function **exit()** is used to exit the system.

Name

facts(), goals() - display working memories

Synopsis

integer facts()

integer goals()

Description

If there is a current expert object then **facts()** and **goals()** will respectively display its FWM and GWM.

These functions display each element of a working memory. If that element is a fact object, then its string representation is used. However a class object's value will only be displayed if a print operation is defined for its class. This print operation can have any return type, must have zero formal parameters, and can either be named **print()** or **Print()**. If both **print()** and **Print()** are defined, then the first one from the top of the class's definition will be used.

Example

```
class employee {

      integer a, b;

      integer Print()
      {

            printf("%d %d", this->a, this->b);

            return;

      } /*Print*/

} /*employee*/
```

Name

fctns() - lists out all system level functions

Synopsis

integer fctns()

Description

Function **fctns()** lists out all system level functions.

Name

free_form() - set the system's free-form flag

Synopsis

integer free_form(state)

integer state;

Description

Function **free_form()** is used to set the system's free-form flag. If this flag is nonzero, then free-form patterns and facts are accepted by the compiler. Otherwise a warning message is issued each time a free-form fact or pattern is compiled. The free-form flag defaults to a nonzero value.

Function **ops_free_form()** is the compiled version of this function.

Name

gensym(), setgen() - generate symbol functions.

Synopsis

integer gensym()

integer setgen(start)

integer start;

Description

Function **gensym()** is used to generate a symbol with the format:

gen<num>

After each **gensym()** call the system increments the value of <num> starting from a default value of zero. The value of <num> can directly be set by using the function **setgen()**.

Example

```
-->
--> setgen( 10 )
--> printf("%s %s %s\n", gensym(), gensym(), gensym())
gen10 gen11 gen12
-->
-->
```

Name

list() - list system information

Synopsis

integer list(keyword)

char *keyword;

Description

Function **list()** is used to list information about the system. The table given below gives a summary of its keywords.

Name	Description
channel	List out the names of all the channels in the system.
eo	List out the names and states of all Expert Objects.
fctns	List out the function prototypes of all nonclass (top level) functions in the system.
rs	If there is a current expert object, then list out the names of its rule sets.
rule	If there is a current expert object and a current rule set, then list out the names of the rule set's rules.
scope	From the current working environment to the root, list out the local variable names of each block.
stats	List out the current expert object's statistics.
vars	List out the local variable names of the current working environment's block.

Name

load() - load a file of definitions into the system

Synopsis

integer load(path)

char *path;

Description

Function **load()** takes one string parameter that specifies the name of an operating system file.

Example

```
-->
--> load("xmas.ops");
--> load("date.ops");
-->
```

Name

open_dribble(), close_dribble() - record system information

Synopsis

integer open_dribble(file, append)

char *file;

integer append;

integer close_dribble()

Description

Function **ops_open_dribble()** turns on the OPS-2000 dribble feature. When this feature is on, all output by OPS-2000 functions is sent to the specified file name. If **append** is nonzero then the output information (dribble) is appended to the contents of the specified file. If **append** is zero then the file is truncated when it is opened for writing.

Function **close_dribble()** turns off the dribble feature, and closes any opened dribble file.

Notes

The Microsoft Windows version of this function dribbles all of text that is sent to the STDIO window. Consequently, the dribble file is an exact copy of the STDIO window.

The command-line version of this function does not dribble the standard input/output streams. It only dribbles values produced by OPS-2000 specific functions. Consequently, function **printf()** does not dribble its output.

Name

ops_send(), ops_receive() - send and receive EO messages.

Synopsis

integer ops_send(channel, mesg)
char *channel;
char *mesg;

integer ops_receive(channel, mesg, steps)
char *channel;
char *mesg;
integer steps;

Description

Function **ops_send()** sends a message to a channel declared using the **defchannel** declaration.

Function **ops_receive()** receives a message from a channel declared using the **defchannel** declarartion.

Notes

These functions should not be called either directly or indirectly from an expert object.

Name

print() - general print functions

Synopsis

integer print(value)
char value;

integer print(size, value)
integer size;
char *value;

integer print(value)
char *value;

integer print(value)
integer value;

integer print(size, value)
integer size;
integer *value;

integer print(value)
real value;

integer print(size, value)
integer size;
real *value;

integer print(value)
symbol value;

integer print(size, value)
integer size;
symbol *value;

Description

Function **print()** is used to display data objects of a particular type. If this function is given two actual parameters, then the second parameter should be a vector of which the first parameter specifies the number of its elements to be displayed.

Example

```
char one[ 100 ];
double y;

--> print( y );                     //print object's value
--> print( 10, one );               //print out first ten elements
--> print( "hello world" );         //print object's value
```

Name

query() - general query functions

Synopsis

integer query(prompt, result)
char *prompt; /*IN: prompt string*/
char *result; /*OUT: char*/

integer query(prompt, result)
char *prompt; /*IN: prompt string*/
integer *result; /*OUT: integer*/

integer query(prompt, result)
char *prompt; /*IN: prompt string*/
real *result; /*OUT: real*/

integer query(prompt, result)
char *prompt; /*IN: prompt string*/
symbol *result; /*OUT: symbol*/

integer query(prompt, result, size)
char *prompt; /*IN: prompt string*/
char *result; /*OUT: string*/
integer size; /*IN: size of result string*/

Description

Function **query()** is used for general input queries. Its first parameter is a prompt string that is displayed when **query()** is called, the other parameters are its input values.

Example

```
integer int_value;
real real_value;
symbol symbol_value;
char char_value;
char string_value[ 120 ];

--> query("enter integer value => ", &int_value);
```

```
--> query("enter real value => ", &real_value);
--> query("enter symbol value => ", &symbol_value);
--> query("enter character value => ", &char_value);
--> query("enter string value => ", string_value, 120);
```

The string **query()** function takes as input at most **size** - 1 characters.

Name

reset() - function for resetting expert objects

Synopsis

integer reset()

integer reset(path)

char *path;

Description

Resetting an expert object means that its working memories and associated structures are set to their initial states, and the expert object is set to state: **running**. This means that in addition to the initial goal and fact, the only other asserted data objects are from the EO's **deffacts** and **defgoals** declarations. Also all channels that have the EO as their sender are reset.

If **reset()** is given no parameters then:

1. If there is a current EO, then it is reset.
2. If not (1) then all of the system's expert objects are reset.

If **reset()** is given a parameter then it must be an expert object name, which will subsequently be reset.

Name

run() - run an expert object.

Synopsis

integer run(steps)

integer steps;

integer run(eo, steps)

char *eo;

integer steps;

Description

Function **run()** is used to run an expert object. The steps parameter is passed to an expert object's inference engine function which interprets its meaning.

If **run()** has one parameter then

1. If there is a current EO, then it is run.
2. If not (1) then repeatedly run all of the expert objects in the system until they have all stopped.

If **run()** has two parameters, then the specified EO is run with the given steps parameter.

Name

strtosym() - convert a string to a symbol

Synopsis

symbol strtosym(str)

char *str;

Description

Function **strtosym()** is used to convert character strings to system symbols. Once a symbol is created, it will exist until OPS-2000 program termination.

Name

system() - execute an operating system command.

Synopsis

integer system(command_line)

char *command_line;

Description

Function **system()** is used to execute an operating system command. This is identical to the "C" **system()** function.

Example

```
-->
--> system("vi .profile");
--> system("vi mab.ops")
--> system("copy mab.ops mab1.ops");
--> system("del mab1.ops");
--> system("ksh");
--> system("csh");
--> system("sh");
--> system("vi");
--> system("ed");
--> system("emacs autoexec.bat");
-->
```

Name

watch(), unwatch() - system monitoring functions

Synopsis

integer watch(keyword)
char *keyword;

integer unwatch(keyword)
char *keyword;

Description

Function **watch()** is used to turn on the monitoring of certain system operations.

Function **unwatch()** is used to turn off the monitoring of certain system operations.

Name	Monitor Description
all	All types of monitoring.
activations	The activation and deactivation of rules.
facts	The assertion and retraction of working memory elements.
fctns	A function's compilation, calling, or exit.
ie	Inference engine function monitoring.
mesgs	The sending and receiving of messages.
rules	The compilation and destruction of rules.

Standard Input/Output

Keyboard and screen I/O functions.

Name

getchar(), putchar() - single character input/output

Synopsis

integer getchar()

integer putchar(ch)
integer ch;

Description

Function **getchar()** gets a character from the standard input.

Function **putchar()** puts a character to the standard output.

Name

printf(), scanf() - formatted string input/output

Synopsis

integer printf(str ...)
char *str;

integer scanf(str ...)
char *str;

Description

These functions are identical to their "C" counterparts.

File Input/Output

The interpreter environment currently cannot handle file pointers as returned by **fopen()**. This is due to the fact that structures cannot be passed between the C++ interpreter and compiled "C" functions.

Name

close(), open(), read(), write() - file I/O functions

Synopsis

```
integer close ( file )
integer file;

integer open( path, oflag, mode )
char *path;
integer oflag;
integer mode;

integer read( file, buffer, nbyte )
integer file;
char *buffer;
integer nbyte;

integer write( file, buffer, nbyte )
integer file;
char *buffer;
integer nbyte;
```

Description

These functions are identical to their "C" counterparts.

String

String operations.

Name

sprintf(), sscanf() - string input to formatted string input/output

Synopsis

integer sprintf(str1, str2 ...)
char *str1;
char *str2;

integer sscanf(str1, str2 ...)
char *str1;
char *str2;

Description

This functions are identical to their "C" counterparts.

Name

strcpy(), strlen(), strcmp(), strcat() - string operations

Synopsis

```
integer strcpy( to, from )
char *to;
char *from;

integer strcpy( to, from )
char *to;
symbol *from;

integer strlen( str )
char *str;

integer strlen( sym )
symbol sym;

integer strcmp( str1, str2 )
char *str1;
char *str2;

integer strcmp( str, sym )
char *str1;
symbol sym;

integer strcat( str1, str2 )
char *str1;
char *str2;

integer strcat( str, sym )
char *str;
symbol sym;
```

Description

The symbol parameter type is passed to compiled functions as a character string. A symbol, like a string constant, should never be modified.

These functions are almost identical to their "C" counterparts. The sole exceptions being where **integer** is returned instead of **(char *)**.

Inference Engine

The inference engine function is used to control the inferencing process. This function uses a set of system defined functions to perform actions such as the firing and removal of activations. This library contains a wealth of functions that provide information on everything from the size of an EO's primary agenda to the retraction of a particular WME.

On error, these functions return negative values.

Library

EO - Expert Object

PA - Primary Agenda

LA - Local Agenda.

IE - Inference Engine

RS - Rule Set

CS - Conflict Set

MS - Match Set

WM - Working Memory

WME - Working Memory Element

ACT - Activation

The general inference engine function format is:

```
integer <name>( ie_id, steps )
integer ie_id;
integer steps;
{

            ...

}
```

Each expert object has an inference engine function associated with it. When this function is called it is passed a unique inference engine id (ie_id) that identifies to the system what expert object the function was called for. This id is mapped to an expert object for the duration of the inference engine function call.

Name

ie_act_rule_info(), ie_act_rule_stats(), ie_act_nth_ms_id() - rule activation functions

Synopsis

integer ie_act_rule_info(ie_id, rs_index, cs_index, name, summary, type, priority)
integer ie_id; /*IN*/
integer rs_index; /*IN*/
integer cs_index; /*IN*/
char *name; /*OUT*/
char *summary; /*OUT*/
integer *type; /*OUT*/
integer *priority; /*OUT*/

integer ie_act_rule_stats(ie_id, rs_index, cs_index, threshold, use, wm_ms_sz, min_wme_id)
integer ie_id; /*IN*/
integer rs_index; /*IN*/
integer cs_index; /*IN*/
real *threshold; /*OUT*/
integer *use; /*OUT*/
integer *wm_ms_sz; /*OUT*/
integer *min_wme_id; /*OUT*/

integer ie_act_nth_ms_id(ie_id, rs_index, cs_index, n, nth_wme_id)
integer ie_id; /*IN*/
integer rs_index; /*IN*/
integer cs_index; /*IN*/
integer n; /*IN*/
integer *nth_wme_id; /*OUT*/

Description

Function **ie_act_rule_info()** is used to get rule definition information for a particular activation. The values passed by variable **type** currently have the following rule type mappings. If the match set is empty (for example all NOT patterns were used), then variable **min_wme_id** will be set to -1.

Value	Actual Rule Type
1	forward chaining/normal form
2	backward chaining
3	forward chaining/fuzzy
4	forward chaining/confidence factor

Function **ie_act_rule_stats()** is used to get rule activation information.

Function **ie_act_nth_ms_id()** is used to return the WME id for the nth member of an activation's match set. An activation's match set is indexed from zero to n - 1. For example:

```
integer i, wme, ms_ct;

ie_act_rule_stats( ..., &ms_ct, ... );

for (i = 0; i < ms_ct; i++) {
      ie_act_nth_ms_id( ie_id, rs_index, cs_index, i, &wme);
      printf("match set index %d; wme id %d\n", i, wme);
      } /*for*/
```

Name

ie_cs_rs_size(), ie_cs_fire(), ie_cs_remove(), ie_cs_fire_remove() - conflict set operations.

Synopsis

integer ie_cs_size(ie_id, rs_index)
integer ie_id; /*IN*/
integer rs_index; /*IN*/

integer ie_cs_fire(ie_id, rs_index, cs_index)
integer ie_id; /*IN*/
integer rs_index; /*IN*/
integer cs_index; /*IN*/

integer ie_cs_remove(ie_id, rs_index, cs_index)
integer ie_id; /*IN*/
integer rs_index; /*IN*/
integer cs_index; /*IN*/

integer ie_cs_fire_remove(ie_id, rs_index, cs_index)
integer ie_id; /*IN*/
integer rs_index; /*IN*/
integer cs_index; /*IN*/

Description

Function **ie_cs_size()** returns a rule set's conflict set (local agenda) size. This value will always be greater than zero since a rule set that appears in the primary agenda has to have at least one rule activation.

Function **ie_cs_fire()** fires a particular member of the conflict set. This conflict set is stored in the rule set's local agenda.

An activation's position in a local agenda may change when it is fired. Consequently, to atomically fire and remove an activation the **ie_cs_fire_remove()** function should be used.

Function **ie_cs_remove()** is used to remove a particular member of the conflict set.

Function **ie_cs_fire_remove()** is used to atomically fire and remove an activation from a rule set's local agenda.

Name

ie_default() - specify the default inference engine

Synopsis

integer ie_default(fctn_prototype)

char *fctn_prototype;

Description

This function is used to set the default inference engine function.

Example

```
int ops_default_ie(int ie_id, int steps)
{
      int run;

      for (run = 0; ((ie_pa_rs_size(ie_id) > 0) && (steps > 0));
                                                                  steps--) {
            run++;

            if (ie_eo_is_stopped( ie_id ))
                  break;

            if (ie_cs_fire_remove( ie_id, 0, 0 ) < 0)
                  break;
            } /*for*/

      if (ie_watch() != 0)
            printf("%d rules fired.\n", run);

      return;

} /*ops_default_ie*/

user_fctns()
{
      ...

      ie_default("integer ops_ie_default(int, int)");

      ...
} /*user_fctns*/
```

Name

ie_eo_stats() - general IE statistics

Synopsis

```
integer ie_eo_stats(ie_id, name, asserts, retracts,  activations, firings, sent, received)
integer ie_id;                  /*IN*/
char *name;                     /*OUT: expert object name*/
integer *asserts;               /*OUT: number of successful asserts*/
integer *retracts;              /*OUT: number of successful retracts*/
integer *activations;     /*OUT: number of rule activations*/
integer *firings;         /*OUT: number of activations fired*/
integer *sent;                  /*OUT: number of messages sent by EO*/
integer *received;              /*OUT: number of messages recv'd by EO*/
```

Description

Returns statistics on the specified EO. These values are reset when the associated EO is reset.

Name

ie_eo_wm_cmp(), ie_eo_wm_size(), ie_eo_wme_remove() - working memory functions

Synopsis

integer ie_eo_wm_cmp(ie_id, compare)

integer ie_id; **/*IN*/**

integer compare; **/*IN*/**

integer ie_eo_wm_size(ie_id, wm)

integer ie_id; **/*IN*/**

integer wm; **/*IN*/**

integer ie_eo_wme_remove(ie_id, wme_id)

integer ie_id; **/*IN*/**

integer wme_id; **/*IN*/**

Description

Function **ie_eo_wm_cmp()** sets an EO's WME comparison flag. If this flag is set to true, then each new fact asserted into the FWM is first compared to all existing facts in the FWM. If the new fact has an identical value to an existing fact, then it is ignored. The WME comparison flag defaults to true.

Function **ie_eo_wm_size()** returns a working memory's size. If **wm** is zero, then the size of the GWM is returned. If **wm** is one, then the size of the FWM is returned.

Function **ie_eo_wme_remove()** is used to retract a WME. This operation acts like a **retract** statement. A WME's id can be retrieved using **ie_act_nth_ms_id()**.

Name

ie_eo_stop(), ie_eo_is_stopped() - EO current state functions

Synopsis

integer ie_eo_stop(ie_id)
integer ie_id; **/*IN*/**

integer ie_eo_is_stopped(ie_id)
integer ie_id; **/*IN*/**

Description

Function **ie_eo_stop()** sets a particular EO's stop flag to true.

Function **ie_eo_is_stopped()** returns the value of an EO's stop flag.

Name

ie_pa_rs_size(), ie_pa_rs_info(), ie_pa_rs_index_by_sys_id(), ie_pa_rs_index_by_name()
- returns primary agenda information.

Synopsis

integer ie_pa_rs_size(ie_id)
integer ie_id; /*IN*/

integer ie_pa_rs_info(ie_id, rs_index, name, rs_system_id, priority, total_rules)
integer ie_id; /*IN*/
integer rs_index; /*IN*/
char *name; /*OUT*/
integer *rs_sys_id; /*OUT*/
integer *priority; /*OUT*/
integer *total_rules; /*OUT*/

integer ie_pa_rs_index_by_sys_id(ie_id, rs_sys_id)
integer ie_id; /*IN*/
integer rs_sys_id; /*IN*/

integer ie_pa_rs_index_by_name(ie_id, name)
integer ie_id; /*IN*/
char *name; /*IN*/

Description

Function **ie_pa_rs_size()** returns the size of an EO's primary agenda.

Function **ie_pa_rs_info()** returns the rule set information for a particular primary agenda index.

Function **ie_pa_rs_index_by_name()** returns a rule set's primary agenda index. It does this by searching for a rule set's name. This name is returned via **ie_pa_rs_info()**. A value of -1 indicates that the rule set is not in the primary agenda.

Function **ie_pa_rs_index_by_sys_id()** returns a rule set's primary agenda index. It does this by searching for a rule set's system id. This system id is returned via **ie_pa_rs_info()**. A value of -1 indicates that the rule set is not in the primary agenda.

Name

ie_watch() - interpreter inference engine watch flag value

Synopsis

integer ie_watch()

Description

Returns the value of the inference engine watch flag. If this value is zero, then the watch flag isn't set. A nonzero value means that it is set. The inference engine flag is set by function **watch()**, and unset by function **unwatch()**.

Embeddable

This library is composed of functions for the purpose of embedding OPS-2000 within your application. For the most part, functions in this library have identical counterparts appearing in the operating environment library.

Functions in this library can only be called from externally compiled code.

Name

ops_agenda() - display agenda function

Synopsis

integer ops_agenda(path)

char *path;

Description

Function **ops_agenda()** displays a primary or local agenda. If **path** is NULL then it identical to **agenda()**, otherwise it is

agenda(char *).

Name

ops_assert_fact(), ops_assert_goal() - assert string value

Synopsis

integer ops_assert_fact(fact, certainty)

char *fact;

double certainty;

integer ops_assert_goal(goal, certainty)

char *goal;

double certainty;

Description

Function **ops_assert_fact()** asserts a string into the current expert object's FWM. The fact is asserted with the certainty value specified by the **certainty** parameter.

Function **ops_assert_goal()** asserts a string into the current expert object's GWM. The goal is asserted with the certainty value specified by the **certainty** parameter.

Name

ops_clear() - clear an expert object or the entire system.

Synopsis

integer ops_clear(name)

char *name;

Description

If parameter **name** is NULL, this function efficiently clears the system of all user loaded definitions. This function does not delete any generated symbol values. Function **ops_terminate()** used in conjunction with function **ops_init()** can be used to restore the system's original state.

If parameter **name** is nonNULL, it is interpreted as an expert object name. The named expert object is cleared from the system. A current expert object cannot be cleared.

Examples

Clearing the system of user loaded definitions.

```
...

ops_clear();

...
```

Restoring the system to its original state.

```
...

ops_terminate();
ops_init();

...
```

Name

ops_ce() - change working environment

Synopsis

integer ops_ce(path)

char *path;

Description

If **path** is NULL, then **ops_ce()** is identical to **ce()**, otherwise it is identical to **ce(char *)**.

Name

ops_declare() - feeds a string definition to the OPS-2000 compiler

Synopsis

int ops_declare(definition)

char *definition;

Description

This function enters a top level definition into the interpreter. There is no maximum definition length.

The interpreter function **declare()** is an interpreter interface to this function.

Examples

```
ops_declare("overload print, run, load;");

ops_declare("integer a, b, c;");

ops_declare("defrelation initial_fact () { }");
```

Name

ops_def_fctn() - adds a compiled function to the C++ interpreter

Synopsis

int ops_def_fctn(ptr, fctn_prototype)

int (*ptr)();

char *fctn_prototype;

Description

This function is used to enter compiled functions into the C++ interpreter environment. The list given below summarizes compiled functions.

1. Returns either a real, integer, char, or symbol value.
2. Pointer values cannot be explicitly returned.
3. Single dimensional arrays of the fundamental types can be passed.
4. Any returned symbol values are entered into the system's symbol table. Symbols are almost identical to string constants, except the equality operators can be applied to them and they should never be explicitly destroyed.

Examples

```
ops_def_fctn(printf, "integer printf(string ...)");

ops_def_fctn(ops_declare, "integer declare( string ) ");
```

Name

ops_default_ie() - the system's default inference engine function

Synopsis

integer ops_default_ie(ie_id, steps)

int ie_id;

int steps;

Description

Function **ops_default_ie()** is the system's default inference engine function. This function should never be directly called. This function should only be used in expert object **ie** declarations and with **ops_def_fctn()**.

Examples

```
int user_fctns()
{

      ...

      ops_def_fctn(ops_default_ie, "int my_default_ie(int, int)");
      ie_default("int my_default_ie(int, int)");

      ...

} /*user_fctns*/

/*
 * Example
 */
defeo Example
{
      ie = int ops_default_ie(int, int);

      ...

} /*Example*/
```

Name

ops_exec() - executing a command-line

Synopsis

integer ops_exec(command)

char *command;

Description

Function **ops_exec()** feeds a string of text to the command-line interpreter.

Examples

```
ops_exec("declare(\"int x;\"); x = 20; print(x + 20);");

ops_exec("printf( \"Hello World!\\n\"); ");
```

Name

ops_facts(), ops_goals() - display working memories

Synopsis

integer ops_facts()

integer ops_goals()

Description

Function **ops_facts()** is identical to function **facts()**.

Function **ops_goals()** is identical to function **goals()**.

Name

ops_free_form() - set's the free-form compiler flag.

Synopsis

int ops_free_form(mode)

int mode;

Description

This is the compiled version of **free_form()**. Its purpose is to set the compiler's free-form flag. If this flag is nonzero then free-form facts and patterns are allowed (default). If this flag is set to zero then a warning message is generated when a free-form fact or pattern is compiled.

Name

ops_init() - initialize the OPS-2000 system

Synopsis

integer ops_init()

Description

This function is for initializing the OPS-2000 system. When embedding OPS-2000 within your system, it must be the first OPS-2000 function call made. This function can only be called again if it is preceded by a call to function **ops_terminate()**.

Name

ops_load() - loads a file of definitions

Synopsis

integer ops_load()

Description

Function **ops_load()** loads a file of OPS-2000 definitions into the system. If an error occurs, then all definitions previously loaded will remain in the system. Consequently, in the event of an error during a load, the entire system must be cleared using function **ops_clear()**.

Name

ops_open_dribble(), ops_close_dribble() - record system information

Synopsis

integer ops_open_dribble(file, append)

char *file;

integer append;

integer ops_close_dribble()

Description

Function **ops_open_dribble()** is identical to function **open_dribble()**.

Function **ops_close_dribble()** is identical to function **close_dribble()**.

Notes

What is written to the dribble file is version dependent. When using Microsoft Windows, all STDIO window output is written to the dribble file. When using non-Microsoft Windows versions, only the output from OPS-2000 specific functions is written to the dribble file. In particular this means that function **printf()** writes to Windows dribble files but not to the other OPS-2000 versions dribble files.

Name

ops_reset() - function for resetting expert objects

Synopsis

integer ops_reset(path)

integer *path;

Description

Function **ops_reset()** is for resetting expert objects. If the value of path is NULL, then it is identical to function **reset()**, otherwise it is **reset(char *)**.

Name

ops_run(), ops_run_eo() - run expert objects

Synopsis

integer ops_run(steps)
integer steps;

integer ops_run_eo(eo, steps)
char *eo;
integer steps;

Description

Function **ops_run()** is identical to **ops_run(integer)**.

Function **ops_run_eo()** is identical to function **ops_run(char *, integer)**.

Name

ops_send(), ops_receive() - send and receive EO messages.

Synopsis

integer ops_send(channel, mesg)
char *channel;
char *mesg;

integer ops_receive(channel, mesg, steps)
char *channel;
char *mesg;
integer steps;

Description

Function **ops_send()** sends a message to a channel declared using the **defchannel** declaration.

Function **ops_receive()** receives a message from a channel declared using the **defchannel** declarartion.

Notes

These functions should not be called either directly or indirectly from a running expert object.

Name

ops_terminate() - terminates the OPS-2000 system

Synopsis

integer ops_terminate()

Description

This function deinitializes the OPS-2000 system. It returns the state of the system to that of before the previous **ops_init()**. This function destroys all symbols created by the last instance of the system.

Function **ops_init()** can be called again to reinitialize the system.

Example

```
int main()
{
    int i;

    /* This loop will run the system 10 times. */
    for (i = 0; i < 10; i = i + 1) {
        ops_init();

            ... perform some actions ...

         ops_terminate();
         }   /*for*/

} /*main*/
```

Name

ops_watch(), ops_unwatch() - system monitoring functions

Synopsis

integer ops_watch(keyword)
char *keyword;

integer ops_unwatch(keyword)
char *keyword;

Description

Function **ops_watch()** is identical to function **watch()**.
Function **ops_unwatch()** is identical to function **unwatch()**.

Math

Mathematical functions accessible from the interpreted and OPS-2000 compiled forms.

Name

ops_rand(), ops_srand() - random number generation

Synopsis

int ops_rand(x)

int x;

void ops_srand(x)

int x;

Description

Function **ops_rand()** generates a random integer number in the range 0 to x - 1.

Function **ops_srand()** can be called at any time to reset the random number generator to a random starting point. The generator is initially seeded with a value of 1.

Name

exp(), log(), log10(), pow(), sqrt() - exponential, logarithm, power, square root functions.

Synopsis

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;

Description

Function	Description
exp()	Returns **e** to the xth power.
log()	Returns the natural logarithm of x. The value of x must be positive.
log10()	Returns the logarithm base ten of x. The value of x must be positive.
pow()	Returns x to yth power. If x is zero, y must be positive. If x is negative, y must be an integer.
sqrt()	Returns the non-negative square root of x. The value of x may not be negative.

Name

floor(), ceil(), fmod(), fabs() - floor, ceiling, remainder, absolute value functions.

Synopsis

double floor(x)
double x;

double ceil(x)
double x;

double fmod(x, y)
double x;

double fabs(x)
double x;

Description

Function	Description
floor()	Returns the largest integer not greater than x.
ceil()	Returns the smallest integer not less than x.
fmod()	Returns the floating-point remainder of the division of x by y.
fabs()	Returns the absolute value of x.

Name

sin(), cos(), tan(), asin(), acos(), atan(), atan2() - trigonometric functions

Synopsis

double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(y, x)
double x, y;

Description

Functions **sin()**, **cos()**, and **tan()** return respectively the sine, cosine and tangent of their argument, x, measured in radians.

Function	Description
asin()	Returns the arcsine of x in the range -PI/2 to PI/2
acos()	Returns the arccosine of x in the range 0 to PI
atan()	Returns the arctangent of x in the range -PI/2 to PI/2
atan2()	Returns the arctangent of y/x, in the range PI to -PI, using the signs of both arguments to determine the quadrant of the return value.

Name

sinh(), cosh(), tanh() - hyperbolic functions

Synopsis

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;

Description

Functions **sinh()**, **cosh()**, and **tanh()** return respectively the hyperbolic sine, cosine, and tangent of their argument.