The FIRE Manual

FIRE Version 3.0

Kenneth D. Forbus
Tom Hinrichs
Johan de Kleer
Madeline Usher
Matt Klenk
Andrew Lovett
Praveen Paritosh

Qualitative Reasoning Group
Northwestern University

Version of 1/3/2017 3:37 PM
Please send feedback and suggestions to
forbus@northwestern.edu

1 Introduction 3
2 Overview of FIRE 3
2.1 FIRE's Knowledge Base 3
2.2 Reasoners 5
2.3 Reasoning in FIRE 5
2.4 Analogical processing in FIRE 8
2.5 Contexts in FIRE 9
3 Getting started 10
3.1 Testing your FIRE installation or port 12
3.2 Starting it up 12
3.3 Shutting it down 13
3.4 Setting up your environment during development 13
3.5 Queries 13
4 FIRE subsystems and APIs 14
4.1 Reasoners 14
4.1.1 Reasoner API 14
4.2 The Working Memory 16
4.3 Knowledge Base API 16
4.3.1 Setting up KB's 16
4.3.2 Storing and retrieving knowledge 17
4.3.3 Deleting knowledge 18
4.4 Loading knowledge from files 18
4.4.1 Structural queries 20
4.5 ASK 21
4.5.1 Effort and Advice 22
4.6 Analogy operations 24
4.6.1 Analogical matching 25
4.6.2 Working with Cases 28
4.6.3 Built-in Case Constructors 29
4.6.4 Analogical retrieval 30
4.6.5 Analogical generalization 32
4.7 QUERY 35
4.7.1 Controlling backchaining 36
4.7.2 Using backchaining effectively 36
4.8 SOLVE 37
4.8.1 Representing suggestions 39
4.8.2 Customizing node cost strategies 41
4.8.3 Debugging tools for solve 42
4.9 The HTN Planner 42
5 Extending FIRE 44
5.1 Adding a new reasoning source 44
5.1.1 Deciding what predicates should be provided 44
5.1.2 Implementing the predicates 45
5.2 Adding new quantifiers 47
6 Extras 47
6.1 Human-Readable Namestrings 47

7 Tips, Tricks, Traps, and Troubleshooting 50
7.1 Problems while getting set up 50
7.2 Problems found during shakedown 50
7.3 Problems during development 51
8 References 51
9 Appendix A: Vocabulary of specialized predicates in ASK and QUERY 53
9.1 Predicates handled specially by ASK 53
9.1.1 Structural knowledge from the KB 53
9.1.2 Structural predicates 53
9.1.3 Metaknowledge predicates 56
9.1.4 The Eval subsystem 57
9.1.5 Dynamic Update Predicates 58
9.1.6 Binding List Predicates 58
9.2 Predicates handled specially by QUERY 58
9.2.1 N-ary predication 59
9.2.2 Metaknowledge predicates handled by QUERY 59
10 Index 60

1 Introduction

The FIRE reasoning engine is designed to support building general-purpose reasoning systems operating over large knowledge bases. Here are the key features of FIRE's design:

- FIRE is a *federated architecture* where *reasoning sources* are used to provide access to specialized facilities, such as spatial reasoners. Thus aspects of inference that are best handled procedurally can be integrated smoothly with other kinds of reasoning. Truth-maintenance services provide a uniform layer for drilling down into underlying assumptions and for generating explanations.
- FIRE knowledge bases are implemented via a persistent object-oriented database, rather than being part of a binary image. This facilitates scaling up, persistent storage, and portability.
- FIRE is designed from the ground up to support analogical processing. That is, support for analogical mapping (via the Structure-Mapping Engine, SME), similarity-based retrieval (via MAC/FAC) and analogical generalization (via SAGE) are built into the software from the beginning. For example, drawing an analogy in FIRE is considered to be more primitive than backchaining. This inverts the usual place that analogical processing is given in reasoning systems, where, if it is included at all, analogy is treated as an extra, optional add-on that might be invoked when all else fails.
- FIRE uses contexts to partition backchaining. Efficient reasoning over large knowledge bases is still, in general, an unsolved problem. A common problem with backchaining in reasoners is that it easily "gets lost" when working in a very large KB. FIRE exploits the microtheory structure of the knowledge base to control which Horn clauses are visible for backchaining. This enables fine-grained declarative control over which axioms are used.

FIRE has been created in collaboration with PARC, Inc. The Plan B database that we use to host the knowledge base was created by Ken Forbus and Johan de Kleer, using Franz Inc.'s AllegroCache database as the persistent object store.

2 Overview of FIRE

This section provides a conceptual introduction to FIRE's facilities, which will provide the foundation needed for understanding the details. We discuss FIRE's knowledge base, the idea of reasoners, and reasoning mechanisms in turn.

2.1 FIRE's Knowledge Base

The knowledge base provides persistent storage of ground facts and axioms. It supports pattern-directed retrieval of facts, and provides rapid inference facilities for structural queries (e.g., types of arguments for predicates, collection membership). The knowledge bases used with FIRE tend to be reasonably large, ~10^6 facts. Given that many of our

current experiments involve learning, the KB infrastructure is designed to scale up to several orders of magnitude beyond that.

FIRE KBs use a Cyc-style microtheory structure to provide contextualization. That is, every fact is stored in one or more microtheories. Most FIRE operations are carried out with respect to some microtheory. Since microtheories can inherit from each other (via the **`genlMt`** relationship), a microtheory in effect specifies a logical environment for the reasoning being conducted.

The rest of this subsection goes into why we use a persistent object database for the KB infrastructure, and can be skipped unless one is interested.

FIRE's predecessor, DTE, was our group's first large knowledge based reasoning engine. DTE stood for "Domain Theory Environment", and was built as part of the DARPA HPKB program. One important advance in DTE was the use of a standard database to store the contents of a KB and to support pattern-matching. Previous attempts to do this either restricted the expressive power of the representation language (e.g., binary relations in [1]) or stored a subset of the information in the database (e.g. storing a few properties of a concept in a database and the rest as a string that had to be loaded when the knowledge was to be used in [15]). Tom Mostek figured out an encoding that was fully general and reasonably efficient, and thus we were able to use Microsoft Access to store DTE's knowledge bases.

Unfortunately, there were some significant problems with DTE's KB infrastructure. First, while Microsoft Access had great tools, the transaction testing, security facilities, and ODBC calls caused considerable performance hits over what a simple flat-file database could provide[1]. Our next solution was to use a streamlined flat-file database (BDL) built by Bob Cheslow of PARC, who has adapted it for our purposes. Second, the pattern to database table encoding used in DTE wasn't quite as efficient as it might be, and the pattern directed retrieval facilities were fairly simple and not optimized. The DBEX system created by Reinhard Stolle[2] and John Everett[3], with consultation from us, provided a more efficient encoding scheme and quite powerful pattern matching facilities. To make structural queries efficient, a *structural cache* was pre-computed and stored with the KB to speed up taxonomic queries. Data that would be inconvenient or inefficient to store inside the assertion database itself was stored in a *resources* directory associated with the KB (e.g., the binary files associated with sketches, including jpeg backgrounds).

The DBEX solution worked much better, but still had its problems. First, BDL really wasn't designed to operate at this scale. This led to significant brittleness. Second, while DBEX was fast some of the time, some of the time it ended up being quite slow. It did not directly support microtheories, for example, and hence the number of database queries performed was linear in the number of microtheories in the current logical

[1] Jim Hendler warned us about this, and he was absolutely right.
[2] Then at PARC, now at BMW.
[3] Then at PARC, now at DARPA.

environment. Third, the structural cache, while vital for efficiency, was itself something of a memory hog (40MB slugged into the Lisp heap, which for 32 bit machines was problematic) and brittle, since it was saved as a special-purpose binary file.

To overcome these problems, Forbus & de Kleer started from scratch, using AllegroCache as a persistent object store. Structural facts (e.g., argument constraints, arity, genls, genlPreds, and genlMt statements) were translated into pointers between specialized objects that represented predicates, collections, and entities. General facts are indexed via a coarse-coded bucket index scheme. Contextualization is handled via an ATMS-like label, which provides essentially constant-time context filtering. This knowledge base infrastructure, optimistically called "Plan B", is what this version of FIRE uses.

2.2 Reasoners

The working state associated with a system using FIRE is stored in one or more *reasoners*. Reasoners include a *working memory* that stores the assumptions and results specific to a particular session or use of an application (as distinct from the knowledge base, which persists across sessions). Thus reasoners constitute a locus of activity in FIRE-based systems. Some applications use a single reasoner (e.g., the Case Mapper system for helping cognitive scientists experiment with analogy). Other applications use multiple reasoners, as a way of keeping different contexts straight (e.g., each sketch in a nuSketch system includes a separate reasoner that holds the results of the visual and conceptual processing on it).

The working memory in FIRE is implemented using an industrial-strength version of the LTRE from Chapter 10 of [3]. It includes two significant enhancements, both due to John Everett, a discrimination-tree index for data and rule retrieval, and fact garbage collection in the LTMS [2].

2.3 Reasoning in FIRE

The only way to build powerful general-purpose efficient reasoners seems to be to organize their operation into layers, where primitive operations are used first to find quick answers, and more complex and extensive reasoning is tightly controlled via reflection. This section describes how FIRE's reasoning systems are layered and intended to be used.

The most primitive, low-level query mechanism is **`ask`**. Given a query, **`ask`** returns answers to that query. The number and form of the answers is determined by keyword arguments to **`ask`**, which are described in Section 4.5.1. Other keyword arguments determine what context the information used is to be drawn from, and qualitatively distinct levels of effort. Contexts are especially useful in working with cases. Effort constraints are useful when a sub-query should be restricted to accessing working memory only, or only doing KB lookup.

All results from ask are fully justified in the Working Memory's LTMS. For example, results derived knowledge base lookup are justified via an assumption of the form

(inKB *<fact>***)**

that is added as part of **ask**'s processing. This provides a form of caching, since `ask` can be restricted to looking for answers only in the working memory, for efficiency. FIRE also supports procedural attachments, directly implementing specialized kinds of reasoning in code. These are called *outsourced predicates*. One class of outsourced predicates are *structural* predicates, concerning the structural properties of particular facts (see Section 9.1.2). Another capability built into **ask** is the ability to evaluate expressions, via the **evaluate** predicate (see Section 9.1.4).

FIRE developers can extend the capabilities of ask via adding *reasoning sources* to a reasoner. A reasoning source provides procedural attachments that handle specific queries, based on the predicate involved and what parameters in the query pattern do or do not contain variables that need to be bound. This allows, for example, authors of spatial reasoning systems to treat a query which asks whether or not a given object is above another one or not quite differently from finding all objects which are above a given object.

Reasoning sources can be quite simple or complex, depending on the functionality provided. Analogical processing is implemented through a reasoning source, which brokers the translation between working memory assertions and SME's internal data structures and the reasoning involved in dynamic case construction. Reasoning systems can also provide interfaces to external resources: For example, in FIRE's predecessor, DTE, a reasoning source was used to provide interoperability to the ArcInfo geographic information system, so that its computational facilities could be harnessed via reasoner queries.

Reasoning sources are stored with reasoners because they often are implemented using specialized software that maintains considerable state (e.g., spatial reasoners, geographic information systems, analogical processing software) and this state must be directly tied to the correct working memory.

ask is designed to be a primitive operation, not involving reflection on the part of the reasoning system. Therefore what **ask** does when answering a query is presumed to be reasonably bounded. "Reasonably bounded" does not always mean fast: For example, asking for a comparison of two large analogs of a thousand propositions each takes time. But those implementing reasoning sources are supposed to do the best they can to ensure reasonable performance, given the complexity of what they are doing. In some cases, this might involve decomposing a complex operation into a number of smaller, tightly constrained queries which are driven by one of the more reflective layers of the system (e.g., **solve** and the HTN planner), so that effective control can be maintained.

The next level of query mechanism is backchaining, accessed via **`query`**. While the KB can contain general-purpose axioms in any form whatsoever, the only axioms used in backchaining are Horn clauses, whose form is

`(<== <consequence> <antecedent1> …<antecedentN>)`

General axioms can be converted into Horn clauses, of course, but FIRE deliberately does not do that automatically. The reason is tractability: A general axiom can be used in a wide variety of ways, but which ways are actually relevant and efficient will vary according to task. It is assumed that whatever is using FIRE (either a system designer or a system that is controlling its own reasoning) will ensure that the appropriate Horn clauses will be available.

A query must always be made with respect to a microtheory, which is a subset of the knowledge base that constitutes a set of relevant knowledge for some task. As noted above, microtheories can inherit from other microtheories via the `genlMt` relation, and hence the starting microtheory and those it inherits from constitute the logical environment for the query. Only Horn clauses accessible within the current logical environment are used in backchaining. Similarly, only ground facts accessible within the current logical environment are used in backchaining. This puts the burden of partitioning axioms and facts onto the microtheory mechanism, which, since it is declarative, opens the door for learning systems to restructure their own knowledge for more efficient inference. For instance, a common practice is to use a microtheory to contain the set of axioms relevant to a particular form of reasoning (which may be one large microtheory, or a small microtheory that inherits from others, for modularity) and another microtheory to provide the background data needed for an inference (which, again, can be one large microtheory or a whole set of them). FIRE's mechanisms are designed to handle microtheories efficiently in the KB, so do not be shy about using them[4].

The third layer in FIRE is a reflective mechanism, consisting of an agenda which manipulates an and/or graph of goals and tasks. It is invoked via a call to **`solve`**. In keeping with the layered structure of FIRE, **`solve`** starts by using **`query`** to see if the result is immediately derivable. (Recall that **`query`** starts by calling **`ask`**.) If it is not, `query` is called to find *suggestions* for how to proceed. Suggestions are basically declarative fragments of knowledge that specify problem solving strategies. The initial goal given to **`solve`** constitutes the root node of the and/or graph. Each suggestion is instantiated as an OR subnode of a goal node, and are queued on the agenda. When a suggestion on the agenda is processed, any subgoals representing information that it requires are created as AND subnodes of the suggestion node, and themselves are queued

[4] Microtheories in working memory are not as well supported at this writing, but that is something that we plan to fix in the future.

on the agenda. This process proceeds until either an answer to the goal node is found, or resource bounds are reached. Items on the agenda can be processed in any order[5].

The **`solve`** mechanism produces only one result by default because it is the deliberative layer of the reasoning system. It is the part that is charged with determining whether or not a given solution will be suitable and going back for more solutions if not. This means that the and/or graph must incrementally produce new solutions, since a solution that satisfies one parent goal might not, for example, satisfy another.

The fourth layer of FIRE is a Hierarchical Task Network (HTN) planner. It reduces high-level, non-operational tasks to linear sequences of primitive operations. The methods for achieving tasks are stored in the knowledge base as plans that expand a high-level task to lower-level primitive tasks. The different methods have preconditions for determining their applicability and preference rules for selecting among alternatives. The planner searches among these methods and simulates the effects of primitives until it finds an action sequence that successfully achieves the task.

The HTN planner provides FIRE-based systems with a means for taking actions. The planner is very general, and relies on **`query`** for the reasoning involved in making its decisions. Plan execution systems, on the other hand, tend to be application-specific.

2.4 Analogical processing in FIRE

Two of our research hypotheses about how common sense works are (1) it relies heavily on within-domain analogical reasoning, and (2) abstract generalizations emerge from a *progressive alignment* process that uses repeated analogical comparisons [13]. This motivated building in analogical processing into FIRE in terms of its basic capabilities[6].

The interaction with analogical processing facilities is carried out by using queries expressed in the *analogy ontology*, a vocabulary of entities and predicates that reifies the concepts of structure-mapping theory [12]. An overview of the concepts in it and how they are used can be found in [11]. The details have changed here and there – for example, constraints on matches are now expressed as part of a match query itself, rather than being specified implicitly in the context surrounding the query, as the 2000 paper describes. See Section 4.6.1 for details.

Analogical matching is carried out using SME, the Structure-Mapping Engine [3,4]. Each match between a base and target results in the creation of an SME object to represent the match, which is returned as one of the variables bound by the query. Subsequent queries about this object are be used to extract the mappings and their correspondences and candidate inferences.

---

[5] We plan to provide a declarative mechanism for specifying preferences, analogous to the goal preference statements supported by SOAR, so that FIRE developers can provide pragma information and FIRE applications can tune their behavior via machine learning.

[6] Analogical processing is implemented as a reasoning source, but every FIRE reasoner includes it by default.

Similarity-based retrieval is carried out using MAC/FAC [8]. The contents of case libraries are specified declaratively. Content vectors are computed on demand, so the first retrieval will include the overhead for computing the content vectors for the descriptions that constitute the case library.

Cases are specified via terms in queries. Cases can be stored explicitly, or constructed dynamically based on the contents of the knowledge base and working memory. The kind of method used to construct the case depends on the functor in the term specifying the case. FIRE developers can hook in their own case construction methods by extending a generic procedure with new methods. These methods can use **`ask`** and **`query`** freely.

Analogical generalization is carried out via SAGE, a descendant of SEQL [16]. SAGE maintains a set of *generalization contexts*, one per concept. Each generalization context takes as input a stream of examples of that concept. Each example is either assimilated into a generalization, if it is sufficiently similar to it, or kept around to start a new generalization if another example arrives that is sufficiently close to it. The generalizations are probabilistic, with frequency information computed for each relationship and attribute based on the number of occurrences in examples assimilated into the generalization.

Generalization contexts can also be used as case libraries for retrieval with MAC/FAC. Generalization contexts can be stored in the knowledge base, providing a persistent learning mechanism for analogical models. Generalization contexts can also be created in working memory, an innovation motivated by recent psychological results indicating that generalization processes are also operating to support rapid learning [14]

2.5 Contexts in FIRE

FIRE uses the conventions of CycL, the representation language used in the Cyc knowledge base, developed by Cycorp. Consequently, we have adopted a version of their microtheory mechanism as a means of describing contexts and cases. Specifically,

- All queries are done with respect to some microtheory. If unspecified, it typically defaults to **`EverythingPSC`**, that is, the global microtheory. This is generally not what you want (see below).
- The logical environment for a query, which specifies what facts can be drawn upon in reasoning, is defined to be the current microtheory plus all of the microtheories that are **`genlMt`**s of it. (As in Cyc, **`BaseKB`** and **`UniversalVocabularyMt`** are typically a **`genlMt`** of every microtheory, but they don't have to be.)
- New statements that are derived are installed in the current microtheory.
- The context can be open, i.e., a non-ground term, such as a variable. In that case, **`ask`** treats the query as a pure retrieval query, subject to the constraints in the effort specification. (The reasons for not doing inference on open contexts are that most of the results would be incorrect given that many contexts are incompatible, it would be very inefficient, and there would be no easily determined place to store the results.)

- When a query can be answered by a reasoning source, the handler for the reasoning source is executed in the logical environment provided by the query.

In assertions, we indicate membership in microtheories via **ist-Information** statements. That is,

(ist-Information *<context>* *<fact>*) indicates that ***<fact>*** holds in ***<context>***.

These **ist-Information** statements are stored explicitly in the WM. The only exception is statements in **BaseKB**; by convention, no **ist-Information** statement is used when stating facts that are in **BaseKB**. Moreover, some predicates are *global predicates*, that is, they are true independent of microtheory. For example, **genlMt** is a global predicate[7]. Structural predicates, i.e. those that are true about the form of a fact itself, also tend to be global predicates. Statements involving global predicates are never wrapped in ist-Information, i.e. they are treated like they are part of **BaseKB**.

Every FIRE KB has at least the following microtheories:

- **BaseKB** contains global facts, things that are true everywhere, such as arity.
- **UniversalVocabularyMt** is very similar.
- **EverythingPSC** is the microtheory that contains all other microtheories. **EverythingPSC** is logically inconsistent, and hence should be used with extreme caution in reasoning. It is useful only for looking up facts when finding out what microtheory would best be used for some task.
- **NothingPSC** is the microtheory that can never contain any facts. Useful to provide a default that does not cause errors, but will force someone to use something more sensible if they want answers.

3 Getting started

FIRE requires Allegro Common Lisp, version 8.1 or higher, either for Windows or for Linux. It may work on other ACL platforms, but we have only used it routinely on these. Much of the code is standard Common Lisp, but AllegroCache is crucial, since it is used to implement the knowledge base.

The source code tree is contained in a Zip archive. You'll need to unpack it into a directory on a hard drive with at least 6 gigabytes free (mostly for the KB's). We recommend using as the root for the tree **<drive>:\qrg**. Edit your **startup.cl** file to load **qrgsetup.lsp**, and in that startup file bind the qrg path (i.e., the value of qrg:*qrg-path*) to wherever you have unpacked everything.

You will need to create a case sensitive (i.e., Modern) image. Whether or not you use an ASCII character set image is your choice, the knowledge base should no longer require

[7] If you think about the possible consequences of **genlMt** statements that are themselves in contexts conditioned on other genlMt statements, you can see that allowing **genlMt** statements to be contextualized could become problematic.

that. If you choose to create an ASCII image, the **qrg\utils** directory has a file that describes step by step how to do this. (We have converted to case-sensitive code in QRG to facilitate interfacing with other software and to live more harmoniously with Cyc predicate naming conventions.)

Please make sure you have downloaded the latest ACL patches, and installed AllegroCache. To install the current version of AllegroCache, type

```
(require :update)
(system.update:install-allegrocache)
```

which will download the latest version from Franz, Inc. There are hard-coded version numbers in the file **planb\defsys.lsp**; you may need to update these if the current version is later than when those files were written.

To load and compile the code the first time, first make sure to load `qrg/qrgsetup.lsp`. This will define the loading functions used by the defsystems. You will probably want to load **qrgsetup** automatically by adding it to the startup.cl file in your acl installation directory. Now load and compile FIRE by executing:

```
(compile-sys "fire" :fire)
```

Then kill that lisp, start again, and just call:

```
(require-module "fire" :fire)
```

in the future. If you get source code updates or have extended the system yourself, periodically repeating the above procedure will ensure that everything is running compiled, which is crucial for efficiency.

You will commonly want to use Rbrowse and Zgraph as well. Rbrowse is a web-based system for inspecting knowledge bases and reasoning. You can load it via

```
(require-module "rbrowse" :rbrowse)
```

Rbrowse supports browsing knowledge bases and the working memories of reasoners. When ontologizing or writing axioms, it is wise to have a KB browser window open so that you can more easily check for what predicates and collections are available and their conventions for use.

Zgraph is our graph plotting and display system. It only works under Windows currently, since it relies on ACL's Common Graphics. You can load it via

```
(require-module "zgraph" :zgraph)
```

Either of these systems can be compiled by using compile-sys, i.e.,

```
(compile-sys <path> <module-name>),
```

where `module-name` is a keyword, such as `:rbrowse` or `:zgraph`. You can always force a system to be reloaded by passing in the keyword argument :force-load to require-module.

An additional note concerning Zgraph: As it happens, the problem of laying out graphs is extremely complex. There is no single best algorithm, so Zgraph incorporates several layout algorithms from the literature as well as a novel one developed by our group. Some of them rely on a program from ATT called WinGraphviz. If you check under zgraph\v9_resources, you'll find a .msi file that will install WinGraphviz on your machine, if you don't have it already.

3.1 Testing your FIRE installation or port

Ours is not a perfect world, and so it behooves one to test software once it has been installed, and especially so for research software. Here is a test you can do to ensure that your FIRE installation is operating appropriately. This tests determines if the basic database mechanisms and analogy mechanisms are up and running.

1. Ensure you are in the cl-user package at the top level. (Franz defaults to cg-user, which is suboptimal from our perspective since we are using cl-user as our data package currently, because it simplifies development and debugging.)
2. Compile and load FIRE, as per the previous section.
3. Run the procedure **`(shakedown-fire)`**.

If you see any warnings, errors reported, or (heaven forbid!) a Lisp error, then there is something wrong. Please see Section 7 for help with specific problems. Otherwise, everything is working normally.

N.B. These regression tests include the construction of a very lightweight KB, the "smoke test KB". This includes a very minimal subset of OpenCyc knowledge files (to test structural inferences) and classic analogy examples (to test matching and retrieval). KBs are stored in **`qrg/planb/kbs`**, so this directory must be writable by the Lisp.

3.2 Starting it up

Once you've loaded FIRE for a programming session, you need to create a knowledge base. There are two aspects to this:

1. Creating a knowledge base from scratch using "flat files" that describe the KB contents.
2. Creating a knowledge base in the lisp environment you are working in that refers to a previously created knowledge base on disk.

Creating a KB from flat files rarely needs to be done; that's the point of having a persistent shared general-purpose KB. Generally you will be, in essence, opening up a KB by creating a lisp object that serves as a connection to the underlying database system. To create a kb use

`(fire::make-fire-kb <kb path>)`

`<kb-path>` is a pointer to the directory where the KB is housed. For example, if your QRG directory is c:\qrg, then the path for the OpenCyc-derived KB is typically **`c:\qrg\planb\kbs\opencyc-kb\`**. While the KB is open, you will not want to do move, write, or copy operations to that directory – doing so can lead to an undefined state and break the KB. The global variable **`fire::*kb*`** points to the currently open KB.

Recall that reasoners are the locus of activity in FIRE. To create a reasoner, use this call:

`(fire::make-reasoner <string for title>)`

There is a global variable, **`fire::*reasoner*`**, which is used by many procedures to implicitly specify the reasoner. The cleanest way to set this variable is via this procedure:

`(fire::in-reasoner <reasoner>)`

Alternately, to specify the reasoner to be used by default during the execution of a piece of code, this macro can be used in your source code:

`(fire::with-reasoner <reasoner> . <code>)`

It is best if you encapsulate these calls as part of the code you are developing, so that you don't have to do this from scratch every time.

3.3 Shutting it down

It is very important that you close the KB before exiting Lisp, by calling

```
(fire::close-kb)
```

If you don't, it can corrupt the indices of the underlying database which leads to very annoying errors. When designing FIRE-based applications, we strongly recommend adding an error handler for desperate circumstances where the Lisp image is going down that closes the KB. There is no close operation on reasoners, since these will be GC'ed by the Lisp environment themselves when they are no longer in use.

3.4 Setting up your environment during development

We follow the usual Lisp programming practice of keeping some global variables around in the environment that serve as "registers" for our working state. Two of the most important ones obviously are

```
fire::*kb*
fire::*reasoner*
```

which are set up automatically when you call the procedures for making a KB and for making a reasoner, respectively. As you become familiar with the API, you will find others that are useful as well.

3.5 Queries

There are several macros and procedures designed for interactive use of FIRE. We summarize a few of the most useful ones here, with more detail appearing in the API information in the appendix.

```
(fire:ask-it <query> &key (reasoner fire::*reasoner*
                           (context :all)
                           (number :all)
                           (facts :all)
                           (env t)
                           (transitive nil)
                           (infer t)
                           (response :pattern))
```

fire::ask has a large number of parameters to provide a decent amount of control over reasoning. When interacting, one often just wants sensible defaults. **fire::ask-it** is a procedure that provides this. Note that **fire::ask-it** does evaluate the **<query>** argument, since backquoted substitutions into a pattern are common.

```
(fire::q <query> &key (context :all)
                      (number :all)
                      (depth 10)
                      (response :bindings)
```

```
(reasoner *reasoner*))
```
invokes backchaining. The results are saved in the following global variables for easy access:

cl-user::*results*	List of results returned by query.

We use the **rbrowse** system as one means of inspecting results.

```
(explain-fact <fact>)
```

provides a simple interface to explore the reasons underlying a conclusion. Similarly,

```
(browse-kb)
```

points your browser to a page for exploring the current knowledge base contents. For looking at results of analogical matches,

```
(browse-current-sme)
```
displays the details of the last match done in the environment, while

4 FIRE subsystems and APIs

This section describes FIRE's subsystems in detail, including their APIs and principles of operation.

4.1 Reasoners

Reasoners are the main locus of processing in FIRE. They contain a Working Memory (described in Section 4.2) which is private to that reasoner, and represents the declarative component of its state. They have a pointer to the knowledge base they are using, which is typically shared between multiple reasoners. They have a set of reasoning sources, representing the procedural attachments associated with specific predicates. Since reasoning sources often have significant internal state, instances of reasoning sources are also private to particular reasoners. Reasoners always have at least one reasoning source, an analogy reasoning source. Analogy is considered to be fundamental to FIRE's operation.

4.1.1 Reasoner API

```
(make-reasoner <title> &key (kb *kb*) (ltre-debug-flags nil)
                                      (type 'reasoner)
                                      (analogy-source-type 'analogy-source))
```
Creates a reasoner whose title is the string <*title*>. The knowledge base defaults to the global KB. It is useful for some applications to specialize the reasoner object and the analogy source object, hence the keyword argument specifying them. The LTRE debugging flags are passed into the working memory of the reasoner when it is created.

reasoner
This global variable provides a register for accessing a reasoner associated with the current task or module.

(in-reasoner *<reasoner>*)
Re-binds the value of ***reasoner*** globally. Use with caution.

(with-reasoner *<reasoner>* . *<code>*)
This macro lambda-binds ***reasoner*** for the scope of the execution of **<code>**, with appropriate protection for non-local exits.

(add-source <source> <reasoner>)
Adds the reasoning source **<source>** to reasoner **<reasoner>**. The following procedures are to be used in procedures that add a source to a reasoner:

```
(register-ask-source ((reasoner reasoner) (functor symbol)
                                  (source source)
                         handler signature result-signature effort-type
                                  &key (ignore-cache? nil)
                                       (type 'source-registry-ask-entry))
```

Installs the procedure **<handler>** as a way to do a query on the predicate **<functor>**, given the argument signature **<signature>**. **<result-signature>** describes the bindings it provides values for, and **<effort-type>** is a heuristic indication of how hard it will be to carry out. The **ignore-cache?** keyword tells FIRE to ignore any cached values when constructing results; this is handy for dynamic predicates.

```
(register-tell-source ((reasoner reasoner) (functor symbol)
                        (source source) handler signature
                         &key (type 'source-registry-tell-entry))
```

Installs the procedure ***<handler>*** as something to call whenever statements whose functor is ***<functor>*** are introduced to the working memory via **tell**.

```
(register-untell-source ((reasoner reasoner) (functor symbol)
                        (source source) handler signature
                         &key (type 'source-registry-untell-entry))
```

Installs the procedure ***<handler>*** as something to call whenever assumptions whose functor is ***<functor>*** retracted working memory via un**tell**.

Some of the information inferred in a reasoner may include structural properties of entities that are also mentioned in the KB. To ensure that one is using both the structural information in the reasoner's working memory as well as in the KB, the following procedures are useful:

```
(collections-of (entity &key (kb *kb*)
                             (reasoner *reasoner*)
                             (context :all))
```

Returns the list of collections of which <*entity*> is known to be a member. The keyword argument context provides a means of limiting the search to the logical environment implied by a given microtheory.

`(instance-of? (entity col (reasoner reasoner))`
Returns non-nil exactly when **`<entity>`** is an instance of **`<col>`** in **`<reasoner>`**.

4.2 The Working Memory

The working memory is built using a Logic-based Tiny Rule Engine (LTRE). It uses the LTRE code in [5] as a starting point, so we strongly recommend reading the appropriate chapters to find out in detail how it works if you need to know that. For scaling up, a discrimination tree is used for indexing, and fact-level garbage collection [2] is supported. The classic procedures for inspecting beliefs and the reasons underlying them (e.g., `fetch`, `fetch-trues`, `why?`, `assumptions-of`, `informant-of`, and the like) are available.

While the LTRE rule system is also available, with a few exceptions, we strongly advise against using it. The only exceptions are when there is a very low-level, very automatic kind of response needed to external events. For example, an LTRE rule is used in nuSketch applications to do certain updates when a glyph is moved or changed. The problem with using LTRE rules is that they are incompatible with systems whose focus of reasoning changes radically over time, or which must continue to operate for long periods (days, weeks, or months) since rule instances have indefinite extent. The other reasoning facilities (`ask`, `query`, and `solve`) provide more powerful and more scalable mechanisms than LTRE rules, and should be used instead.

Procedures that are useful when working with FIRE's working memory include

- **`(ltre::facts-which-mention <obj> &optional (ltre *ltre*) (believed-only? t))`** returns the list of working memory facts that mention **`<obj>`** within **`<ltre>`**. The LTMS label is used to filter those which are not currently believed if the **`believed-only?`** flag is non-**`nil`**.

4.3 Knowledge Base API

4.3.1 Setting up KB's

- **`(make-fire-kb <file path>)`** creates a new KB in the directory denoted by <*file path*>.
- **`(open-kb <file path>)`** establishes a connection with the KB stored at <*file path*>, returning a handle to the KB.
- **`(in-kb <kb>)`** binds the default KB (**`*KB*`**) to <*kb*>.
- **`(with-kb <kb> . <expressions>)`** binds the default KB to <*kb*> and evaluates <*expressions*> in that environment.

- **(close-kb <*kb*>)** closes <*kb*>, that is, it closes the underlying database associated with it. Subsequent calls to KB operations will result in errors.

In the rest of the API procedures, the optional keyword argument `:KB` denotes the KB to use. If not supplied, it defaults to ***KB***.

4.3.2 Storing and retrieving knowledge

- **(kb-store <*expression*> &key (kb *kb*) (mt nil))** Stores <*expression*> in <*kb*> with microtheory <*mt*>. <*mt*> must be provided – it will error out if not.
- **(retrieve-it <*pattern*> &key**
  (:number <*default nil, integer or* :ALL>)
  (*:response <:pattern or :bindings*>)
  (:context nil))
  retrieves axioms matching <*pattern*> in <*kb*>. The number of items returned depends on the :number keyword argument. If context is specified, retrieval is restricted to that context.
- **(retrieve-all <*pattern*>)** like RETRIEVE, but with :number = :ALL.
- **(retrieve-references <*exp*> &key *mt*)** retrieves all axioms that contain <*exp*> as a subexpression. If <*mt*> is non-nil, retrieval will be restricted to this particular microtheory. This procedure is intended for use in interfaces, browsers, and debugging, not inner loops of reasoning systems.
- **(retrieve-nat-references <*nat*> &key mt)** is like retrieve-references, but retrieves all axioms that mention the non-atomic term **<*nat*>**.

The following procedures are useful for inspecting microtheories and their contents. Some of them use microtheory objects, not forms, which can be found via **kb::retrieve-microtheory**, whose **:create?** argument controls whether the object is created if not already in existence.

- **(immediate-genlmts <mt>)** returns a list of all of the microtheories that **<mt>** inherits from directly.
- **(all-genlmts <mt>)** returns a list of all of the microtheories that **<mt>** inherits from, i.e. transitively.
- **immediate-specmts**, **all-specmts**, are like the above, but in the reverse direction.
- **(list-facts-in-mt <mt obj>)** returns a list of fact objects (not forms) for all of the facts in **<mt obj>**, a microtheory object
- **(map-over-facts-in-mt <procedure> <mt obj>)** calls **<procedure>** over each fact object (not form) explicitly stored in **<mt obj>**.
- **(count-facts-in-mt <mt obj>)** returns the number of facts directly stored in **<mt obj>**.
- **(tabulate-mt-sizes)** returns an alist whose entries have as their key a microtheory name and as their value, the size of that microtheory.

For example, in the smoke test KB built via the FIRE regression test,

```
cl-user(11): (kb::count-facts-in-mt (kb::retrieve-microtheory 'base-5))
40 ;; This should always be the same value
cl-user(12): (apply '+ (mapcar 'cdr (kb::tabulate-mt-sizes)))
6084 ;; This will change as the regression test evolves
cl-user(13): (length (kb::tabulate-mt-sizes))
141 ;; Ditto
```

The Plan B KB has a journaling facility built-in. Please see **planb\journaling.lsp** if you need to use this. It is dusty.

4.3.3 Deleting knowledge

- **(forget-fact <fact> <mt>)** causes **<fact>** to be removed from **<mt>** in the KB.
- **(forget-mt <mt>)** causes all facts in a microtheory in the KB to be forgotten. All **genlMt** statements involving that microtheory will remain intact (since **genlMt** statements are global, i.e. not contextualized).
- **(nuke-kb-item <entity>)** removes the conceptual entity **<entity>** and all of the facts which refer to it. If **<entity>** is a microtheory, all of the facts in that microtheory are removed, as are all **genlMt** statements involving it.

4.4 Loading knowledge from files

Knowledge level programming is still programming, albeit at a higher level. When adding knowledge by hand, the usual medium of files to store information and text editors to manipulate them is used, rather than trying to do everything through a specialized software environment. This is useful for three reasons:

1. It allows knowledge bases to be reconstructed from scratch. This is important when there have been a lot of KB changes by many people, or when the underlying FIRE software changes in an incompatible way.
2. It is closest to the standard workflow in programming, which reduces entry barriers.
3. It allow archiving of learned knowledge and results produced by FIRE-based systems in forms that can easily be analyzed.

We call such files *flat-files*, since they are textual precursors to what, inside a FIRE KB and working memory, will be structured representations. The usual file extension for these is "**meld**", but other extensions will work as well. Lisp format for assertions is assumed.

Flat files are loaded by the following procedure

- **(meld-file->kb *<file>*)** loads ***<file>*** into the currently open knowledge base.

Most expressions are interpreted directly as assertions to be stored in the knowledge base. The following expressions are treated specially:

- **(in-microtheory *<mt>*)** indicates that the expressions which follow should be loaded into microtheory ***<mt>***. ***<mt>*** is created if it does not always exist. There can be multiple in-microtheory statements in a flat file, although this is relatively rare, for reasons explained below.
- **(defSuggestion . *<contents>*)** indicates that ***<contents>*** is a SOLVE suggestion (see Section 4.8.1). The loader expands these statements into multiple assertions, as explained later.

Here is an example of a very simple flat file:

```
;; A simple flat file
(in-microtheory NorthwesternUDataMt)
(isa NUSmartClassroom Collection)
(genls NUSmartClassroom Classroom)
(relationAllExists contains NUSmartClassroom DataProjector)
;; End of flat file
```

For efficiency, the database objects constructed for collections, predicates, and microtheories are distinct. They cannot be overloaded. Consequently, the loading process actually runs through a file several times, first extracting information that would impact the internal type system, and then adding regular facts on top of that. If you see an error of the form

<constant> **cannot be** ***<type>,*** **already** ***<other type>***

Then you have accidentally overloaded ***<constant>***. You will have to figure out where this problem is happening (using a combination of the KB browser and string search over the flat files typically works well) and use **kb::nuke-kb-item** to wipe out that constant, and reload the necessary flat files.

This raises an important difference between knowledge-level programming and traditional programming. Expressions mentioning a predicate contribute to its meaning, rather than there being a single definition of it. When one reloads a file with procedure definitions, they replace the old definitions. When one reloads a flat file, the new information is added to the old information. This can be very disconcerting when first encountered. Appropriate use of microtheories can greatly simplify one's workflow. A common practice is to limit the contents of a flat file to a single microtheory, and for simple systems, put all knowledge in that microtheory into that file. The process of updating a microtheory consists of forgetting the current contents (see **forget-mt** in Section 4.3.3) followed by reloading the flat file. Of course, for large bodies of knowledge (e.g. the contents of FrameNet) this is too unwieldly, so defining load procedures that first clear the contents of a microtheory (or those involved in a subsystem of knowledge) and then reload a set of flat files is commonly done.
Another important practice is the convention of *spindle microtheories*. A spindle microtheory is a microtheory that collects knowledge from others, and makes it available for some purpose. There is nothing different about a spindle microtheory per se, it is a matter of convention. Spindles (in the KB) can be nested without loss of efficiency, so the improved modularity for debugging makes them almost always worthwhile.

Importantly, only two packages should ever be used in flat files:

- The **data** package is a synonym for **cl-user**. The vast majority of symbols live in this package.
- Symbols from the **keyword** package are sometimes used as template variables for assertions concerned with natural language processing, providing guides for substitution (e.g. **:agent**, **:object**).

No other package qualifiers should be used in flat files.

4.4.1 Structural queries

Structural queries provide rapid, reflexive inference, based on structural information (**isa**, **genls**, **arg*N*isa**'s and so on). Inheritance is the only supported inference mechanism in these queries. The intent is that these queries are cheap, and used to help suggest/weed out possibilities for simple questions and as subroutines used in more complex reasoning.

In structural queries, when there is a positive result the second argument is a list of axioms that are the antecedents which support the answer. This information is intended to be used in TMS justifications and in the construction of argument structure. These queries all return nil to indicate a false value.

The structural queries include:

- **(instance-of? *<entity>* *<collection>*)** returns **t** iff **(isa *<entity>* *<collection>*)** is either explicitly known in the KB or is derivable from inheritance. No other reasoning is allowed.
- **(subset-of? *<collection1>* *<collection2>*)** returns **t** iff **(genls *<collection1>* *<collection2>*)** is either explicitly known or is derivable via inheritance. No other reasoning is allowed.
- **(arity *<predicate>*)** returns an integer indicating the arity of predicate <*predicate*>, unless the predicate is n-ary. If the predicate is n-ary, the value is the symbol **:n-ary**.
- **(arg-isa *<predicate>* *<integer or* :all*>*)** If the second argument is an integer, returns the collection that the corresponding argument to <*predicate*> must be. (N.B. if the argument is not a member of that collection, then in the case of an attribute or relation the statements must be false. For functions, it means that the result is undefined.) If the second argument is the symbol **:all**, then a list of collections is returned, one for each argument position. In the case of n-ary predicates or functions, the list returned contains a single element, the collection that all arguments must be members of.
- **(result-isa *<function>*)** returns the collection that serves as the range of the function <*function*>. (N.B. we are assuming that overloading, if it were to be used, does not change the range. Otherwise, we would have to specify the collection signature of the arguments as well to specify the range.)
- **(n-instances-of *<col>* &key (kb *kb*) (reasoner *reasoner*)**
  (context EverythingPSC)
  (stop-number -1))
  returns the number of instances of ***<col>*** within ***<context>*** found both within the KB

and in working memory. If **<stop-number>** is other than **-1**, this procedure will return the number of instances up to **<stop-number>**, for circumstances where one only cares that there are at least *n* instances accessible from some context. When **<context>** is **:all**, all microtheories are used. If you want to only look at working memory or the KB, the subroutines **n-instances-of-in-kb** (with required second argument of the KB) and **n-instances-of-in-wm** (with required second argument of the reasoner) carry out those duties, with the other keyword arguments being the same.

- **(n-instances-of-transitive <col> &key (kb *kb*)**
  (reasoner *reasoner*)
  (context EverythingPSC)
  (stop-number -1))

  is exactly like **n-instances-of**, except that it conducts its tally to include all subcollections of <col>.
- **(n-statements-of <pred> &key (kb *kb*)**
  (reasoner *reasoner*)
  (context EverythingPSC)
  (stop-number -1))

  is analogous to **n-instances-of**, except for top-level statements involving the predicate **<pred>**.
- **(n-statements-of-transitive <pred> &key (kb *kb*)**
  (reasoner *reasoner*)
  (context EverythingPSC)
  (stop-number -1))

  is exactly like **n-statements-of**, except that its tally includes all top-level statements involving specPreds of **<pred>**.

4.5 ASK

Ask provides the lowest-level, "reflexive" reasoning facility. Because of the large number of arguments to **ask**, it is assumed that applications will call **ask-it** in order to use keyword arguments.

(ask-it <query> &key <reasoner> <context>
<number> <coverage> <facts> <env>
<transitive> <infer> <response>)

The arguments to **ask-it** have the following meanings:

- <*query*> is the pattern that is being asked about.
- <*reasoner*> is the FIRE reasoner which the query is put to.
- <*context*> is the case or microtheory about which the query is made. Default is **:all**, meaning the global environment.
- <*number*> is either **:all**, indicating all solutions should be found, or a positive integer, indicating the number of solutions desired.
- <*response*> controls the form of solutions returned. **:bindings** indicates that the list of variable bindings for each solution should be returned. **:pattern** indicates that the returned solutions should be the original pattern, with variable

substitutions made. Otherwise, the value is treated as a new pattern that will be returned, with appropriate substitutions, for each solution.

- The remaining arguments indicate how much work the system should do. This is described in Section 4.5.1 below.

ask works roughly as follows:

1. Check for already-known facts in the working memory
2. Check the KB
3. If the predicate is a specialized predicate (e.g., structural), use special-purpose methods to handle it.
4. Check sources to see if they can handle it.

When given a context to work in, ask first looks for information in that local context, and then checks with the global environment (i.e., the WM contents and the **BaseKB** microtheory in the knowledge base).

4.5.1 Effort and Advice

There are four basic controls pertaining to the amount of effort that **ask** will expend in carrying out a query.

1. *Work only in local context, or use the full logical environment?* Generally one wants to use the full logical environment, as described in the section on contexts. However, sometimes it is important to focus on one specific context by itself.
2. *Use inference as well as looking up facts, or just look up facts?* Generally one wants to use the inference supplied by sources as well as looking up facts. However, sometimes it is useful to examine what is already known, and restricting the system to lookup allows one to do that.
3. *When looking up facts, use just the working memory, just the KB, or both?* Generally one wants to use both, but sometimes when reasoning about the state of the system's knowledge or processing, it can be useful to restrict consideration to just WM or just the KB.
4. *When reasoning about collections where open variables are involved, should solutions be generated by walking up (or down, as appropriate) the lattice of the appropriate relationships, or only generate answers corresponding to a single level?* For queries where there are no open variables, i.e., **(isa *<constant1>* *<constant2>*)**, **(genls *<constant1>* *<constant2>*)**, and **(genlPreds *<constant1>* *<constant2>*)**, lattice-walking is always used since we are looking for a single yes-no answer. But if one of the arguments for these predicates is an open variable, lattice-walking can lead to huge numbers of solutions, few of which are actually relevant.

There are two ways to describe effort:

1. Declaratively, as part of a query. This uses special predicates, *ask advice predicates*, that describe control information. Ask advice predicates are treated specially, in that they are never stored as top-level statements in working memory. This special treatment is accorded because we want to split "how" from "what" in WM

justifications. Clauses whose terms use ask advice predicates will still be found in the justification, to support FIRE-based systems reasoning about how to modify their own operation via changing the advice used on particular terms.

2. Procedurally, in calls to **`ask`** and **`ask-it`**. This is for FIRE developers who need fine-grained control over the inference machinery in **`ask`**.

We describe each in turn.

4.5.1.1 Ask Advice Predicates

Here are the Ask predicates that are currently supported, grouped according to functionality.

Controlling whether full logical environment is used:

- **`(localOnly <query>)`** <*term*> is to be solved using only information in the current context.
- **`(contextEnvAllowed <query>)`** The entire logical context specified by the current context can be used when solving for <*query*>.

Controlling whether queries involve inference or retrieval:

- **`(lookupOnly <query>)`** Only fact lookups will be carried out in solving for <*query*>.
- **`(inferenceAllowed <query>)`** Inference via reasoning sources will be allowed in addition to fact lookup when solving for <*query*>.
- **`(nonTransitiveInference <query>)`** Open variables in <*query*> that are arguments to the structural predicates isa, genls, and genlPreds will be solved only by looking at the specific facts known in the appropriate context(s).
- **`(useTransitiveInference <query>)`** Open variables in <*query*> that are arguments to the structural predicates isa, genls, and genlPreds will be solved by finding all of the possibilities involved by walking the lattice of the appropriate relationship(s).

Controlling where facts for queries are sought:

- **`(wmOnly <query>)`** Only working memory can be accessed to look for facts to be used in solving <*query*>)
- **`(kbOnly <query>)`** Only the KB can be accessed to look for facts to be used in solving <*query*>.
- **`(outsourcedOnly <query>)`** Consult outsourced handlers only, ignoring the KB and working memory cache
- **`(allFactsAllowed <query>)`** Facts from both the working memory and the KB can be used in solving <*query*>.
- **`(exactMatchOnly <query>)`** Require retrieved facts to exactly match the query. No unification is performed.
- **`(groundOnly <query>)`** Only return ground statement (no variables in results).

Controlling effort:

- **(numAnswers <number> *<query>*)** Specify the number of answers sought. Equivalent to binding the :number argument to ask.
- **(cacheComplete *<query>*)** Cut off KB retrieval and backchaining inference if results are found in working memory.
- **(ignoreTimestamps *<query>*)** Return all results, including cached results justified by stale timestamps.
- **(honorTimestamps *<query>*)** Filter out cached results justified by stale timestamps.
- **(withTimeout <seconds> *<query>*)** Terminate the query after <n> seconds, whether or not it has completed.
- **(withBackchainingDepth <depth> *<query>*)** Binds the maximum backchaining depth to ***<depth>***

Controlling abduction:

- **(withAbduction <query>)** allows abductive assumptions to be made while proving ***<query>.***
- **(withAbductivePredicates <predicate list> <query>)** enables statements involving predicates on <predicate-list> to be assumed as needed when trying to show **<query>.**
- **(withAbductivePolicy <policy> <query>)** uses abductive policy **<policy>** when trying to prove **<query>.** **<policy>** can either be **MinimalAssumptions**, **AllAssumptions**, **NoAssumptions**, or a procedure corresponding to a scoring policy to maximize.
- **(withCounterfactual <asn> <query>)** attempts to prove **<query>** while temporarily assuming **<asn>.** Justifications for **<query>** are created by discharging their dependence on **<asn>.**

For controlling analogy operations, the following query wrappers are the most common:

- **(reverseCIsAllowed *<query>*)** For analogical matches, compute candidate inferences in both directions.
- **(useMinimalAscension <query>)** For analogical matching, allow minimal ascension to find non-identical relation matches.
- **(noMinimalAscension <query>)** For analogical matching, don't use minimal ascension to look for non-identical relation matches.
- **(entitySupportedInferencesAllowed <query>), (entitySupportedInferencesNotAllowed <query>)** control whether entity correspondences are sufficient to enable candidate inferences.

4.6 Analogy operations

Analogy operations are invoked as queries to **ask**, but they are so central to FIRE's operation that we summarize them here.

4.6.1 Analogical matching

Analogical matching is implemented via SME[8]. To compare two descriptions, the predicate

(matchBetween *<base>* *<target>* *<constraints>* *<match>*)

requires that <*base*>, <*target*>, and <*constraints*> be provided as input, with <*match*> being open and hence produced as output when **ask** is called. <*base*> and <*target*> are either microtheories or terms denoting cases. Microtheories used when the contents of the case are already explicitly known, e.g. a theory, model, or episodic memory might be stored as the contents of a microtheory. Terms denoting cases are used for *dynamic case construction* [18], where facts satisfying the criteria specified by the term are gathered and treated as a case. Dynamic case construction is useful for comparing concepts or entities. For example,

```
(fire:ask-it '(matchBetween solar-system rutherford-atom (TheSet) ?m))
```

When evaluated with respect to the smoke test KB produces the classic solar system/Rutherford atom analogy[9], while

```
(fire:ask-it '(matchBetween (CaseFn Cat) (CaseFn Dog) (TheSet) ?m))
```

compares the concept of Cat with the concept of Dog.

There is a translation process applied to convert CycL statements into SME's standard form. In particular, statements of the form

(isa *<entity>* *<collection>*)

are automatically translated into attribute statements inside SME, i.e.,

(*<collection>* *<entity>*)

These statements are automatically translated back into isa statements at the level of FIRE, although when browsing the SME internals, you will see attribute statements instead of **isa** statements.

<*constraints*> is a set of constraints that are applied to the match. The supported set of constraints is:

- **(requiredCorrespondence *<b_i>* *<t_j>*)** indicates that base item ***<b_i>*** must match target item <t_j> in every mapping created for this match.
- **(excludedCorrespondence *<b_i>* *<t_j>*)** indicates that base item ***<b_i>*** must not match target item ***<t_j>*** in any mapping created for this match.
- **(identicalFunctions)** indicates that non-identical function matches, which SME normally permits when suggested by a larger matching structure, are

[8] The version of SME used in FIRE is the standard portable SME source code, with no FIRE-specific changes. We use the dynamic properties of CLOS to define FIRE KBs as a new kind of SME vocabulary, so that SME-level operations work transparently.

[9] The smoke test KB diverges from CycL conventions because it includes a number of classic analogy examples, including the Karla the Hawk stimulus set [8].

disallowed. This is useful for within-domain reasoning, e.g., solving physics or thermodynamics problems, where the equations apply to quite specific quantities.

- **(excludedCross-PartitionCorrespondences *<set of partitions>*)** indicates that any entity which is a member of one of the collections of a partition (i.e., has that attribute) cannot match with any entity that has an attribute from one of the other partitions. Only explicitly known attribute statements in the base and target are used.
- **(requireWithinPartitionCorrespondences *<partition>*)** indicates that entities for whom one of the attributes in ***<partition>*** holds can only match with entities with one of the same attributes. This is useful in within-domain mappings.

There are three analogy control predicates that are very important for keeping analogical reasoning tractable and appropriate:

- **(notForAnalogy *<thing>*)** means that predicate or function ***<thing>*** indicates a statement or term that should not be included in any analogical reasoning. This is commonly used for bookkeeping information (e.g., **comment**).
- **(ubiquitousForAnalogy *<thing>*)** indicates that statements whose predicate is ***<thing>*** should not be automatically placed into alignment, unless a pair of such statements is part of a larger structure that suggests aligning them. The idea is that there are some kinds of statements that are extremely common in a description (e.g., equations in a physics problem), so common that trying every combination of them would make the size of the match hypothesis forest that SME computes become bloated. If such statements were irrelevant to the purpose of the analogy, they could be filtered out (via **notForAnalogy**), but often such statements are important (again, equations in a physics problem).
- **(atomicAnalogyNat *<function>*)** indicates that non-atomic terms using ***<function>*** should be treated as atomic entities. This is in contrast with SME's usual policy of placing corresponding arguments of non-atomic terms into correspondence. This allows NATs with different functions, or NATs and atomic terms, to correspond.

FIRE evaluates these predicates within the context provided for doing the match using **ask**, so that systems can dynamically adjust them by changing what microtheories are included in the logical environment used for the query.

The open variable provided for <*match*> will be bound to a term which represents the match computed by SME. The form it takes is

(MatcherFn *<id>* *<count>*)

where ***<id>*** is an integer representing how many SMEs have been created in that environment so far, and ***<count>*** indicates the number of times that SME has been updated[10]. The match can be used to access the mappings that SME computed for that base and target. Mappings are reified in FIRE as non-atomic terms as well, e.g.,

(MappingFn *<id>* *<match>*)

[10] SME can operate incrementally, extending the mappings as items are added to the base and target. See [6,7] for details.

where **<id>** is an integer that identifies the mapping with respect to that particular match, denoted by the non-atomic term **<match>**. Correspondences and candidate inferences are reified similarly

The analogy ontology provided with FIRE is based on [11], but it has been extended considerably since then. Here are some of the most useful predicates to serve as a starting point, the rest can be found in **fire/flat-files/analogy-ontology.lsp**:

- **(bestMapping <match> <mapping>)** indicates that **<mapping>** has the highest structural evaluation score of those computed for **<match>**. SME computes up to three mappings by default, if they are sufficiently close to the best one.
- **(baseOfMatch <match> <case>)**, **(targetOfMatch <match> <case>)** indicate that **<case>** is the base or target of **<match>**, respectively.
- **(structuralEvaluationScoreOf <mapping or mh> <number>)** indicates that **<number>** is the structural evaluation score computed for the mapping or correspondence (aka match hypothesis, or MH) **<mapping or mh>.**
- **(numberOfCorrespondences <match or mapping> <number>)** indicates that when **<match or mapping>** is a mapping, then **<number>** is the number of correspondences in the mapping. When **<match or mapping>** is a match, then **<number>** indicates the size of the match hypothesis forest computed during SME's early processing.
- **(correspondsInMapping <mapping> <base item> <target item>)** indicates that base entity or statement **<base item>** corresponds to target entity or statement **<target item>** in mapping **<mapping>.**
- **(candidateInferenceOf <ci> <mapping>)**, **(reverseCandidateInferenceOf <ci> <mapping>)** indicates that candidate inference **<ci>** is a surmise about the target or base suggested by the correspondences of **<mapping>**, respectively.
- **(candidateInferenceContent <ci> <expression>)** indicates that the candidate inference **<ci>** has the propositional content **<expression>**. Note that candidate inferences can include *analogy skolems*, entities conjectured in the base or target in order to satisfy structural consistency. Analogy skolems are reified as non-atomic terms of the form **(AnalogySkolemFn <original>)**, where **<original>** is the entity being projected from the original description. For example, **(AnalogySkolemFn Pressure)** should be read as "something like **Pressure**."
- **(candidateInferenceSupportScore <ci> <number>)** indicates that candidate inference **<ci>** has a support score of **<number>.** The support score is the degree to which the candidate inference is grounded in the mapping, and can range between zero and 1. See [9] for details.
- **(candidateInferenceExtrapolationScore <ci> <number>)** indicates that candidate inference **<ci>** has an extrapolation score of **<number>.** The extrapolation score is the degree to which the candidate inference is conjectural. It can range from 0 to 1, and directly trades off against the support score. See [9] for details.

Minimal ascension is supported in FIRE as a means of relaxing the identicality constraint. The following query wrappers are provided to control this:

- **(noMinimalAscension *<query>*)** means minimal ascension is not used. This is the default.
- **(useMinimalAscension *<query>*)** means minimal ascension should be used.
- **(withMinimalAscensionMultiplier *<weight>* *<query>*)** means that each level moved up the predicate or collection (for attributes) hierarchy needed to find a match, deprecate the score by ***<weight>*.**

4.6.2 Working with Cases

Cases are sets of facts used in analogical processing. Broadly speaking, there are two kinds of cases:

- Stored cases. However they were generated originally, the set of facts in them is typically assumed to be fixed. They might represent the episodic memory of a problem-solving episode, something learned by reading, or background knowledge.
- Dynamic cases. These cases are generated from the contents of the knowledge base for particular purposes. For example, cases might be based on all of the facts that mention an entity, or filtered based on task constraints (e.g. "economic aspects of Germany"). Filtering is often used on stored cases to focus analogical reasoning on a subset of a larger experience, e.g. just the conceptual information in a sketch illustrating a physical principle.

In FIRE, cases are implemented as microtheories. This means that they can be stored persistently in the knowledge base, for example. Recall that the facts available when reasoning in a microtheory include all of the facts which are part of its logical environment, as specified by `genlMt` statements. This is not true for cases: It is only those statements literally in that particular microtheory that are considered to be among the contents of a case.

The arguments to analogy predicates are often just the names of microtheories. However, for dynamic cases, a subset of logical functions, called *case constructors*, are used to denote cases. The first time an analogy operation is carried out on a non-atomic term involving a case constructor, its facts are generated and stored in an SME description object. (Often they are reified in the working memory as well, via an `ist-Information` statement with that case name.) The following predicates support manipulation of cases:

(constructCaseInWM *<case term>*)

<case term> is a non-atomic term involving a case constructor. When used via **`tell`**, it both creates an SME description object in the analogy source in the current reasoner, and reifies the statements in the working memory. This is useful for inspecting cases aside from doing matching, retrieval, or generalization – those predicates automatically construct cases as needed, relying on cached versions otherwise.

(copyWMCase *<from>* *<to>)

(copyWMCaseToKB *<from>* *<to>*)

When asserted via tell, these predicates cause the contents of the case ***<from>*** to be copied to ***<to>***. ***<from>*** must have already been constructed, by using it in an analogy operation or via **constructCaseInWM**. The prior contents of ***<to>*** are not cleared, so that one can accumulate the combination of information from multiple cases via repeated copying. If this is not desired, it is important to forget their contents first.

(caseMentionsEntity *<case term>* *<entity>*)
(caseMentionsPredicate *<case term>* *<predicate>*)

These statements hold if ***<entity>*** or ***<predicate>*** is mentioned in a statement in ***<case term>***. ***<case term>*** must be ground, but the other arguments can be variables, to generate the appropriate answers. Note that the ***<predicate>*** argument includes the higher-order case of the predicate itself being used as an argument to a statement, not just that it is used as the predicate in a case in the statement.

(caseFact *<case term>* *<fact>*)
(numberOfCaseEntities *<case term>* *<#ents>*)
(numberOfCaseFacts *<case term>* *<#facts>*)

The predicate **caseFact** can be used to test if a fact is in ***<case term>***, or as a (less preferred) way to generate the facts in a case. The other two predicates compute the number of entities or facts in a case, respectively.

(matchHypothesisForestEstimate *<base>* *<target>* *<size>*)

This predicate estimates the size ***<size>*** of the match hypothesis forest that SME would construct, if given ***<base>*** and ***<target>*** to match. This is handy for detecting when a match is too large to tackle, and more filtering is needed.

4.6.3 Built-in Case Constructors

FIRE has a number of built-in case constructors that are generally useful.

(ExplicitCaseFn *<mt>*)
(CanonOrderExplicitCaseFn *<mt>*)

These constructors create a case from what is explicitly known in ***<mt>***, in both the working memory and the KB. **CanonOrderExplicitCaseFn** further sorts the facts during the insertion process, to facilitate order-specific regression tests. These put facts into the WM as well as creating an SME description.

(KBCaseFn *<mt>*)
(KBCaseFn-Probability *<mt>*)

These construct cases out of the facts in the knowledge base for microtheory ***<mt>***. They do not include facts in WM, as **ExplicitCaseFn** does. **KBCaseFn-Probability** also filters out facts that have zero probability, which can happen due to SAGE's operation.

(MinimalCaseFn *<term>)*
(MinimalCaseFromMtFn *<term>* *<mt>*)

These constructors gather all of the facts that mention ***<term>*** either in the current logical environment or found explicitly in ***<mt>***, respectively. The analogy control predicates in the current logical environment are respected, e.g. **notForAnalogy** and **atomicAnalogyNat**. Because many microtheories inherit from **BaseKB** and/or **UniversalVocabularyMt**, **MinimalCaseFromMtFn** is often the more useful constructor.

(CaseFn *<term>)*
(CaseFromMtFn *<term>* *<mt>*)

Exactly like their minimal counterparts, except that they also add collection information about the entities in them, which often improves matching.

(AskCaseFn *<query>*)

Constructs a case out of the answers found for ***<query>***. Handles **TheSetOf** explicitly, treating it as an extensional query.

(CaseUnionFn *<set>*)
(CaseIntersectionFn <set>)
(CaseComplementFn <set>)

Creates a case from the union, intersection, or complement of the cases specified by the case terms in the set ***<set>***, respectively.

4.6.4 Analogical retrieval

Analogical retrieval is implemented via MAC/FAC. The basic predicate for invoking retrieval via ask is:

(reminding *<probe>* *<library>* *<constraints>* *<case>* *<match>*)

<probe> is a case, which can be a microtheory or a dynamically specified case, as with matching. ***<constraints>*** are match constraints that will be used by SME during the FAC stage of MAC/FAC. ***<case>*** and ***<match***> are the case retrieved and the SME match from the FAC stage, respectively. Only ***<case>*** and ***<match>*** can be open in reminding queries, the other arguments must be ground terms.

<library> is the case library used. Case libraries are stored in the knowledge base, since they represent a persistent aspect of a reasoner's memory. Whether or not a case is in a case library is controlled by facts of the form

(caseLibraryContains *<library>* *<case>*) indicates that case library ***<library>*** contains case ***<case>***. **caseLibraryContains** is a global predicate, i.e. the contents of a case library are the same, independent of microtheory. It is implemented as a special fact, so that storing and retracting statements of this form in the KB cause the appropriate updates to be made in the persistent KB data structures.

The following modifiers are supported for case libraries:

- **(CaseLibrarySansFn *<library>* *<case>*)** denotes the case library consisting of the contents of the case library ***<library>*** without the case ***<case>***. This is useful when the first reminding is not quite appropriate, and one wants to see the next reminding.
- **(CaseLibraryMinusFn *<library>* *<case set>*)** denotes the case library consisting of the contents of case library ***<library>*** once the cases in ***<case set>*** are removed. This supports searching through memory multiple times.
- **(CaseLibraryUnionFn *<library set>*)** denotes the case library consisting of the union of all of the cases found in the set of case libraries ***<library set>***. This is useful when combining case libraries.

CaseLibrarySansFn and **CaseLibraryMinusFn** are fully compositional. For simplicity of implementation, **CaseLibraryUnionFn** is not: It can appear inside one of the other modifiers, but neither of them can appear in the list of cases inside the set of case libraries, and unions have to be flat, not nested.

Case libraries can contain other case libraries. This is designed to support using SAGE for classification, e.g. the generalization pools for a task (which are themselves case libraries) can be sub-libraries of the case library representing that task. This is the case library analog of spindle microtheories. Storing or retrieving the following assertions makes one case library a sub-library of another:

- **(subCaseLibrary *<sub>* *<super>*)** indicates that case library ***<sub>*** is a sublibrary of case library ***<super>***.

The following predicates are useful for inspecting properties of case libraries:

- **(caseLibraryLocalSize *<lib>* *<n>*)** indicates that there are ***<n>*** cases in case library ***<lib>***, ignoring any contained case libraries.
- **(caseLibrarySize *<lib>* *<n>*)** indicates that case library ***<lib>*** has ***<n>*** cases in it, including all of the contained case libraries.

Because case libraries can be quite large, the internals of retrieval operations are not reified by default. If one is debugging retrieval problems, the global variable ***record-retrievals*** can be set to non-**nil**, which will cause a datastructure per retrieval to be cached with the reasoner's analogy source. Please be cautious in using this facility, since it can easily accumulate enough memory to cause heap blow-outs.

The following Lisp procedures are useful in dealing with case libraries:

- **(kb:list-case-libraries)** generates a list of all case libraries.
- **(kb:map-over-case-objects *<procedure>* *<lib term>*)** applies ***<procedure>*** to each case within ***<lib term>***. ***<procedure>*** must take two arguments, the first the case object, the second the case library itself. The second argument is useful for determining which nested case library a case was found in, e.g. in classification.
- **(kb:map-over-cases *<procedure>* *<lib term>*)** is like **map-over-case-objects**, except that ***<procedure>*** takes only one argument, the lisp form of the case.
- **(kb:show *<lib>* &key (details? t) (stream *standard-output*))** provides a concise printed summary of information about ***<lib>***.

MAC/FAC can exploit multiple cores, when the switch **fire::*use-multicore-macfac*** is non-nil, which is the default. The procedure **(fire::estimate-number-of-cores)** is used to automatically ascertain how many cores a Lisp has.

4.6.5 Analogical generalization

Analogical generalization is carried out with SAGE. SAGE maintains a list of generalizations and examples associated with a concept, which are called a *generalization pools*. Generalization pools are implemented as a subclass of case libraries, so that MAC/FAC can be used on them. Examples are microtheories from the KB, i.e., cases.[11] SAGE operates slightly differently in working memory versus the knowledge base (i.e. long-term memory). We start with the persistent memory version, and then describe the working memory version.

Here are the basic SAGE operations, which are invoked via fire::tell:

- **(sageSelectAndGeneralize *<e>* *<gpool>*)** adds example ***<e>*** to generalization pool ***<gpool>***. The most similar item already in the pool is found via analogical retrieval, using MAC/FAC. If the most similar item is a generalization, the example is added to that generalization. If the most similar item is an example, the examples are merged to form a new generalization. Otherwise, ***<e>*** is added to the set of unassimilated examples. The degree of similarity required is determined by that generalization pool's *assimilation threshold*, which is between 0 and 1. Similarity scores are normalized by the mean of the self-similarity scores for the base and target.

[11] There has been a change in terminology: Generalization pools used to be called "generalization contexts", which caused unfortunate confusions with other notions of context. This has been changed in the API but some of the internal procedures still use the old terms, including data structure definitions (e.g. gpool) for backward compatibility with KBs. Similarly, "example" is now the preferred term, not "exemplar" – the latter means exemplary example, whereas the unassimilated examples in a generalization pool are outliers.

- **(sageSelect <e> <gpool> <reminding> <mapping>)** given an example <e> and a generalization pool **<gpool>**, returns the most similar generalization or outlying example as the binding of **<reminding>**, with **<mapping>** bound to the mapping between them.
- **(SageInstatiateGeneralization <gmt> <bindings> <instMt>)** treats generalization <gmt> as a schema, and adds all of the facts of **<gmt>** to **<instMt>** with the substitutions of **<bindings>.**

It is sometimes more convenient to access SAGE in the course of a query:

- **(sageAddExampleToGpool *<gpool>* *<case>* *<g>*)** with generalization context ***<gpool>*** and case ***<case>*** binds ***<g>*** to the generalization that ***<case>*** was assimilated to, if any. If ***<case>*** was not assimilated, then ***<g>*** will be the case itself.

Often it is clear from context which generalization pool a case should be added to. However, the decision of where to add an example can also be made automatic. Generalization pools can have *entry conditions* that indicate when a case should be added to it. Using the following predicate via tell invokes this mechanism:

- **(sageAddExample *<case>*)** adds ***<case>*** to every generalization context whose entry pattern is satisfied by ***<case>*** and is relevant in the current logical environment. Note that the set of generalization pools is potentially very large, so we do not provide any return value.

Classification is accomplished by the following query:

- **(sageClassify *<case>* *<gpools>* *<gpool>* *<item>* *<sme>*)** uses analogical retrieval with example ***<case>*** as the probe and the union of generalization pools ***<gpools>*** as the case library. The example or generalization ***<item>*** is what MAC/FAC retrieves, with generalization pool ***<gpool>*** being where ***<item>*** came from. The label associated with ***<gpool>*** is thus the classification for the example. The match between the two is also returned, since this can be used to extract a score and also similarities/differences involving them.

Several other predicates are provided for breaking up SAGE operations, either for debugging or for using in larger-scale systems, like Companions, where operations are more distributed. These predicates include **sageGeneralize**, **sageGeneralizeWithMapping**, and **sageAddUngeneralized**. You should use these predicates instead of the above only if you are very familiar with SAGE.

Generalization pools can be inspected with the following predicates:

- **(gpoolExample *<gpool>* *<e>*)** indicates that ***<e>*** is an unassimilated example in generalization pool ***<gpool>***.
- **(gpoolGeneralization *<gpool>* *<g>*)** indicates that ***<g>*** is a generalization in generalization pool ***<gpool>***.

The following predicates allow the parameters of a generalization pool to be inspected or set, depending on whether the second argument is an open variable or a value:

- **(gpoolRelevanceContext *<gpool>* *<mt>*)** indicates that generalization pool ***<gpool>*** is potentially relevant in any logical environment which includes microtheory ***<mt>***, and irrelevant otherwise.
- **(gpoolEntryPattern *<gpool>* *<pattern>*)** means that ***<pattern>*** is the entry pattern for ***<gpool>*.**
- **(gpoolEntryPatternCriterion *<gpool>* *<test>*)** means that ***<test>*** is used to evaluate the entry pattern. The two options are **:ask** and **:contains-pattern. :ask** checks to see if the entry pattern is satisfied by a top-level expression in a candidate example. **:contains-pattern** checks to see if there is a subexpression in any of example's facts that matches the pattern, which is useful if the pattern in question only occurs in the context of a conjunction or a higher-order relation.
- **(gpoolStrategy *<gpool>* *<token>*)** sets the strategy used by SAGE. The possible values are **:gel**, **:bestgel**, **:slowrad**, and **:macfac**.
- **(gpoolUseProbability *<gpool>* *<token>*)** if ***<token>*** is **True** then probabilities are used, and if **False**, then they are not.
- **(gpoolAssimilationThreshold *<gpool>* *<value>*)** sets the assimilation threshold used in ***<gpool>*** to ***<number>*.**

To clean things up:

- **(nukeGpool *<gpool>*)** wipes out ***<gpool>*.**

Generalizations can be inspected with the following queries:

- **(gmtNExamples *<gmt>* *<n>*)** indicates that ***<n>*** examples have been assimilated into generalization ***<gmt>*.**
- **(gmtEntityHistory *<gmt>* *<e>* *<info>*)** indicates that within generalization ***<gmt>***, generalized entity ***<e>*** has historical info ***<info>***. Currently this is a list of the specific entities it was derived from, but this is likely to change in the future, e.g. calculating statistics when there are numerical entity values, setting a limit on how far back the history goes.
- **(sageConstituent *<example>* *<gmt>* *<gpool>*)** indicates that case ***<example>*** is one of the constituents of generalization ***<gmt>***.

For archiving and accumulating knowledge across KB builds, it is useful to save out generalization pools as meld files:

- **(kb::gpool->meld-file *<gpool>* *<file name>*)** constructs a file ***<file name>*** (a fully qualified path, name, and extension) based on the contents of generalization pool ***<gpool>***. This includes its generalizations and examples. Note that, like other meld file load operations, this will be additive: If you load these files into a KB where changes have been made to any of the examples, facts may be added to them. Using **fire::meld-file->kb** should be sufficient to reconstruct the generalization pool.

The Working Memory version of SAGE, which constructs interim generalizations, is very similar. Here is how to set up working memory gpools[12], via **`fire::tell`**:

- **`(setWMGpoolAssimilationThreshold <wmg> <n>)`** sets the assimilation threshold for working memory generalization ***<wmg>*** to ***<n>***, assumed to be a float between 0.0 and 1.0. 0.0 means everything will be assimilated into a single generalization, 1.0 means only if they are identical structure (i.e. the entities could be different).
- **`(setWMGpoolMaxSize <wmg> <n>)`** puts an upper bound of ***<n>*** on the number of items in ***<wmg>***.
- **`(sageWMSelectAndGeneralize <e> <wmg>)`** adds example ***<e>*** to ***<wmg>***.

The following queries provide information about a working memory generalization pool:

- **`(wmGpoolThreshold <wmg> <n>)`** binds ***<n>*** to the assimilation threshold for ***<wmg>***.
- **`(wmGpoolMaxSize <wmg> <n>)`** binds ***<n>*** to the maximum size of ***<wmg>***.
- **`(wmGpoolExample <wmg> <e>)`** indicates that ***<e>*** is an example in ***<wmg>***.
- **`(wmGpoolGeneralization <wmg> <g>)`** indicates that ***<g>*** is a generalization in ***<wmg>***.
- **`(wmSageSelect <c> <wmg> <item> <m>)`** runs the WM select operation (linear search of items, ordered by recency) over ***<wmg>*** with ***<c>*** as the probe, returning the closest example or generalization ***<item>***, and its mapping ***<m>*** indicating how ***<c>*** and ***<item>*** align.

4.7 QUERY

The procedural interface is

```
(query formula &key
                     (reasoner *reasoner*)
                     (context NothingPSC)
                     (number :all)
                     (coverage :specs)
                     (facts :all)
                     (env t)
                     (transitive t)
                     (infer t)
                     (depth 10)
                     (response :bindings))
```

`Query` uses all Horn clauses visible from the logical environment given for the query, via the argument context. (The default argument for context, **`NothingPSC,`** contains no facts, as an inducement to provide something more reasonable.) The depth argument sets the maximum depth bound for inferences. The other keyword parameters are essentially the same as for `ask`.

[12] There is an asymmetry in the APIs for the working memory versus long term memory because in the latter parameters are stored as special facts in datastructures, as opposed to explicit assertions.

The command-line procedure **q** is just like **query**, except that it also sets up a global parameter, ***results***, for convenient debugging.

4.7.1 Controlling backchaining

For tractability, query is restricted to using Horn clauses. This is not as much of a limitation as it might first appear, as the impressive body of systems built in Prolog attests. Many axioms can be turned into a set of Horn clauses that captures most or all of their original meaning (although it might require skolemization if the original axiom involves existential quantification). Recall that the syntax used for Horn clauses is

(<== *<consequent>* . *<antecedents>*)

i.e., using the KIF backchain operator.

A major source of control for backchaining is the use of logical environments in reasoning. Recall that all queries are with respect to a microtheory. Only the Horn clauses accessible within this microtheory will be used in making that query.

Many of the efficiency tricks that work in crafting Horn clauses for efficient reasoning in Prolog are relevant for crafting Horn clauses in FIRE. For example, the ordering of antecedents in a rule can make a substantial difference in terms of performance. However, there are some key differences:

1. FIRE's backchaining does not operate as a generator, like Prolog does. Instead, it produces all answers that satisfy the query (although this can be modified somewhat via the **:number** keyword). Consequently, there is no notion of cut.
2. In Prolog, the order in which clauses are used in a query are determined by the order in which they appear in a program listing. This sequentiality, combined with cut, provides a means of describing conditionals, among other things. In FIRE, since the Horn clauses are all drawn from the current logical environment, there is no way to impose an order on them.
3. Prolog defaults to negation by failure. That is, if something cannot be proven, then it is assumed to not be true. FIRE does not do this by default. Negation by failure can be implemented for particular predicates using **uninferredSentence** if necessary. Enabling the backchainer to use the full notion of negation used in the LTRE is something for a future version.

4.7.2 Using backchaining effectively

- It's typically a good idea to put the axioms for some specific purpose in a separate microtheory, and define the axioms in a separate flat file (or more than one, if there are a lot of them). That way it can be reloaded as often as needed without redefining other predicates (see Section 4.4 for details).
- All Horn clauses are stored in the KB. Horn clauses in WM are ignored. This is for efficiency, and might change in a future version.
- Microtheories provide an extremely useful form of modularity. You can debug sets of axioms independently of each other, and control their composition to do more sophisticated reasoning, simply by making and retracting genlMt

statements. This declarative form of control has been useful in building learning systems on top of FIRE.

- It is important to remember that the reasoning services in FIRE form a hierarchy. **`ask`** may be freely called by **`query`**, but not the other way around. In doing some kinds of complex reasoning, this may mean doing some queries before others, to ensure that information is correctly set up when needed. An exception to this is the evaluatable function **`TheClosedRetrievalSetOf`**, which no longer needs this sort of pre-query. Although technically it is invoked from an ask handler, it can recursively invoke query when called from a query with non-zero stack depth.
- A common mistake in writing Horn clauses is not considering which variables might be open in a query. FIRE allows variables to be bound to other variables,[13] and so the inattentive rule author can produce rules which, in practice, lead to queries with all parameters open. This is typically a bug, e.g. there are over 600,000 **`isa`** statements in some KBs. If you want to enforce that a particular rule has a particular variable in its logical signature bound, the structural predicate **`groundExpression`** should be used as one of the first antecedents to ensure this.

4.8 SOLVE

As described in Section 2.3, **`solve`** is the first level of reasoning in FIRE-based systems that is designed to be interruptible. That is, given their resource limits, **`ask`** and **`query`** will always run to completion; there is no chance of intervening in their operation (at least without dropping into the Lisp code internals). **`solve`** is different. **`solve`** uses an and/or tree to incrementally construct solutions. Even during the process of constructing a single solution, whether or not to proceed is a choice that can be made by external systems, by using flags and by controlling the agenda that **`solve`** uses to carry out its work. There is also a "pause button" that can be used to suspend effort, even if there are more things that could be tried.

The principles underlying **`solve`** are summarized in Chapter 8 of [5]. The key differences between that code and the implementation in FIRE are

- Multiple solutions can be generated incrementally, rather than only providing one solution. This is why it is a tree rather than a graph, as was used in [5]; sharing subgoal nodes does not work when nodes essentially become generators of solutions. Fortunately, the use of an LTMS in WM ensures that relevant work is shared whenever possible.
- Suggestions about how to solve problems are found via **`query`**, rather than antecedent rules. This enables the set of solution plans to be expanded declaratively, which is important for building learning systems.

There are two ways to invoke the **`solve`** subsystem. The first, and simplest, creates an and/or tree and runs it until either the first solution is found or resources are exhausted. The second way is to operate it incrementally, allowing additional processing to be done between each agenda item. Note that, even with incremental operation, **`solve`** uses

[13] Unification in FIRE does an occurs-check, and most unification is done via the method of left/right binding lists.

query to attempt to directly prove each subgoal and to find suggestions, and these uses of **query**, like any other, are considered to be atomic at the level of FIRE operations and hence cannot be interrupted without diving down to the Lisp level.

Here is the simple way of invoking the solve subsystem:

```
(solve <goal> &key (reasoner *reasoner*) (context :all)
                   (response :pattern)
                   (max-depth *solve-max-depth*)
                   (max-agenda-work *solve-max-agenda-work*)
                   (suggestions-mt *default-suggestions-mt*)
                   (single-solution nil)
                   (fail-duplicates nil))
```

This is the non-incremental way of invoking this subsystem. **solve** returns the first solution it finds to ***goal*** (if any) and a pointer to the and/or tree. The keyword arguments ***context***, ***response***, and ***effort*** have the same meanings as they do for **ask** and **query**. ***max-depth*** is the maximum depth for the and/or tree, and ***max-agenda-work*** is the maximum number of agenda items that will be allowed during this run. The ***suggestions-mt*** parameter indicates the microtheory which is used for finding suggestions. **fail-duplicates**, when non-**nil**, invokes the heuristic that if a new goal has the same content as another goal that earlier failed in problem-solving, then that new goal is immediately failed. This is sound only in some domains, and hence is off by default. ***single-solution*** is a similar heuristic: if a node has been solved one way, no further effort to generate solutions for that node is allowed. This, too, is only sound in some domains, and is turned off by default.

(get-solution <*ao-tree*>)

Generates the next solution from the and/or tree ***ao-tree***, if any. This may involve both using alternate bindings found previously and processing new agenda items. The same limits on agenda work and depth specified when the and/or tree was created are still in force. This procedure runs until the next solution is found or it runs out of resources, i.e., it is not incremental in its use of the agenda.

The next four procedures support incremental operation:

```
(setup-solve-aotree <goal>
 &key (reasoner *reasoner*)(context :all)(response :pattern)
      (effort :all)(max-depth 5)(max-agenda-work 100)
      (single-solution nil)(fail-duplicates nil)
      (suggestions-source :KB))
```

returns an and/or tree whose root node goal is ***goal***, but does not attempt to solve it. The arguments are interpreted the same way as in **solve**.

(next-solve-step <*ao-tree*>)

processes the next item on the agenda of ***ao-tree***, if any. It returns two arguments. The first is a flag which takes on one of the following values:

- **:solved** means that the second argument is a solution. The form of the solution is determined by the **:response** parameter given when the and/or tree was created.
- **:failed** means that no more solutions could be found.

- **:agenda-empty** means there is nothing else that can be done.
- **:in-progress** means there are still more agenda items to process, and hence a solution might yet be found.

The second result, the solution, is nil unless the flag returned is **:solved**.

(run-solver-n-steps <*n*> <*ao-tree*> &key (monitor-procedure nil))
Run up to ***n*** items from the agenda of ***ao-tree***, stopping earlier if either a new solution has been found or if resource limitations have been reached. If non-**nil**, the keyword argument ***monitor-procedure*** must be a procedure that will be called after each processing of an agenda item. This procedure will be called with three arguments, **ao-tree**, the flag returned from processing the current agenda item (as defined for **next-solve-step**), and the solution, which is nil if no solution was just found.

(run-to-solution <*ao-tree*> &key (monitor-procedure nil))
runs ***ao-tree*** until the next solution, or failure, or running out of resources.

The following procedures enable work on a particular and/or tree to be paused or unpaused:

(pause-ao-tree <*ao-tree*>)
sets the pause flag for **ao-tree**. All of the procedures above respect the pause flag, and will process no further agenda items once it is set, returning after the current item is fully processed.
(unpause-ao-tree <*ao-tree*>)
resets the pause flag for **ao-tree**, allowing processing of it to continue, if one of the procedures above is used to do so.

4.8.1 Representing suggestions

Ultimately suggestions are translated into axioms in the KB. These axioms are the reference form for suggestions. However, it is useful to have a civilized format that can easily be read by people as well. We will start with the human-readable form, show what axioms it expands into, and then describe some specialized goals recognized by the solve subsystem.

```
(defSuggestion <name> <goal>
     &key documentation test subgoals result-step cost-function)
```

This form can be used in any FIRE flat file, and when loaded via **meld-file->kb**, will be translated into the appropriate assertions. ***name*** is the term used to refer to this suggestion in the KB. ***goal*** is the pattern of problem that it is applicable to. ***subgoals*** are the goals which need to be solved in order to apply this suggestion. (A value must be supplied for **:subgoals**, but all other keywords are optional.) ***result-step*** is a query that is made once the subgoals have been solved, if further computation is needed to construct an answer from them. ***documentation*** is a string that serves as a comment for the suggestion. ***cost-function*** provides goal-specific advice on estimating cost for an instantiated suggestion of this type.

The following predicates are used in defining suggestions in the KB:

(suggestionGoalForm <*name*> <*goal-pattern*>)
indicates that the suggestion ***name*** can be used for goals of the form ***goal-pattern***.
(suggestionSubgoals <*name*> <*subgoals*>)
indicates that the suggestion ***name*** requires the list of ***subgoals*** to be solved.
(suggestionResultStep <*name*> <*result-query*>)
indicates that some of the bindings used in the goal pattern for the suggestion ***name*** are computed by the query pattern ***result-query***.

Here is an example of how a suggestion is transformed into axioms:

```
(defSuggestion VolumeStrategyForCount
     (CountContained ?contained ?container ?count)
     :documentation "strategy for finding the count using volumes"
     :test (and (containsExpression ?criteria
                   (physicallyContains ?container ?contained))
                (hasAttributes ?container Volume)
                (hasAttributes ?contained Volume))
     :subgoals ((volumeOfObject ?container ?vol-container)
                (volumeOfObject ?contained ?vol-contained))
     :result-step (evaluate ?count (QuotientFn
                                    ?vol-container ?vol-contained)))
```

results in the following axioms being placed in the KB:

```
(comment VolumeStrategyForCount
         "strategy for finding the count using volumes")
(suggestionResultStep VolumeStrategyForCount
  (evaluate ?count (QuotientFn ?vol-container ?vol-contained)))
(suggestionSubgoals VolumeStrategyForCount
  (TheList (volumeOfObject ?container ?vol-container)
           (volumeOfObject ?contained ?vol-contained)))
(suggestionGoalForm VolumeStrategyForCount
                    (CountContained ?contained ?container ?count))
(<== (suggestFor (CountContained ?contained ?container ?count)
                 VolumeStrategyForCount)
     (containsExpression ?criteria
                         (physicallyContains ?container ?contained))
     (hasAttributes ?container Volume)
     (hasAttributes ?contained Volume))
(isa VolumeStrategyForCount Suggestion)
```

The context used for storing these assertions is defined by whatever microtheory is in force during the loading of that portion of the flat file. Thus what suggestions are available during a problem-solving task can be controlled based on the logical environment it is performed in.

The following special types of goals are recognized by **solve**:

(solveAll <*binder-form*> <*subgoal-form*>)

is a means of specifying conjunctive goals when you cannot know in advance how many conjuncts there are, because it is data-dependent. ***binder-form*** is a pattern that must bind one or more variables. ***subgoal-form*** must be solved for each of those sets of variable bindings.

For example, in solving an equation,
(solveAll (equationFormulas ?equation ?x ?quantity) (nvalue ?x ?value))
indicates that, for each of the variables in the equation, one must find its numerical value.

(solveSequentially *<var>* *<list>* *<subgoal-pattern>*)
is like **solveAll**, in that it finds a solution for ***<subgoal-pattern>*** with ***<var>*** bound sequentially for each item in ***<list>***. ***<var>*** is assumed to be free in ?subgoal-pattern. Other free variables in ***<subgoal-pattern>*** are uniquified in all but the last subgoal. This enables the last full set of bindings to be passed back, while avoiding conflicting bindings within earlier subgoals. For example, in implementing QP theory, the following solve suggestion handles influence resolution:

```
(defSuggestion ResolveInfluences-Main
    (resolveInfluencesIn ?state)
  :subgoals
  ((findInfluencesInState ?state ?quantities)
   (solveSequentially ?q ?quantities
                      (dsValue ?q ?value))))
```

The predicate **findInfluencesInState** binds **?quantities** to a list of quantities in the state, ordered causally, so that the **solveSequentially** subgoal can then walk through them, finding how quantities are changing in that state.

4.8.2 Customizing node cost strategies

Learning what strategies pay off for different kinds of problems is a classic machine learning issue. Being able to exploit such knowledge requires the cost evaluation mechanism in solve to somewhat flexible. This flexibility is provided by using the following two methods in the computation of costs, e.g.,

(estimate-ao-node-cost <*ao-node*>)
returns a numerical estimate for the cost of and/or node ***ao-node***.

(default-estimate-ao-node-cost <*ao-node*>)
returns a default estimate for the cost of and/or node ***ao-node***.

Both of these methods can be specialized on the type of and/or node. The default method for **estimate-ao-node-cost** simply calls **default-estimate-ao-node-cost**. There are two built-in cases for **default-estimate-ao-node-cost**, one for suggestion nodes and one for goal nodes. These defaults are arranged so that deeper nodes are more costly than shallower nodes, and goals involving more easily evaluated predicates are slightly cheaper. This biases the default operation of **solve** towards shorter, simpler solutions, in a breadth-first fashion.

By subclassing **default-estimate-ao-node-cost**, different default problem-solving strategies can be implemented, e.g., depth-first operation. By subclassing **estimate-ao-node-cost**, knowledge that has been learned about the utility of different methods may be exploited (by using **ask** or **query** to calculate costs, using **evaluate**). When no learned knowledge is available, some default stratagem is still required, hence the separate definition of the two methods.

4.8.3 Debugging tools for solve

A number of useful tools can be found in **fire\v3\tools\solve-debug-utils.lsp**. There is also a zgraph display for AO trees, defined in **fire\v3\graph-types\aotree-graph.lsp** that is extremely handy for debugging. The displays it produces are color-coded, with green meaning that the subgoal succeeded, red meaning that it failed, and grey meaning that it was rendered moot. Another useful zgraph display can be found in **fire\v3\graph-types\suggestions-graph-display.lsp**, which shows the connections between suggestions available from the current microtheory. This is helpful to see what is already there in the current logical environment.

4.9 The HTN Planner

The HTN planner is invoked via

```
(plan <query> &key context reasoner facts
                   depth return-reasons?
                   coverage env transitive infer stack)
```

where

- ***<query>*** is a task specification, of the form **(actionSequence (TheList *<task>* … *<optional other tasks>*))**, where the query is in the **data** package and the task predicates are defined as **ComplexActionPredicates**.
- **context** should be a **PlanSpecificationMicrotheory** that inherits from the domain context containing facts and plans. This context is almost entirely ephemeral. The planner takes an initial set of facts cached in working memory in this context and treats them as the initial state. As it searches for plan expansions, it retracts state facts that are no longer true and justifies new state facts with the new state.
- **facts** tells the planner where to look for plans and static domain facts. Although state facts are all cached in working memory, the plans are generally stored in the knowledge base. The default value of the **:facts** argument is **:kb**. For efficiency reasons, it is often a good idea to pre-cache domain plans and facts in working memory so that the planner doesn't keep returning to the kb for methods.
- **reasoner** is the reasoner the work is performed in.
- **depth** is the maximum expansion depth the planner will use.
- **return-reasons?** produces an explanation for the plan, as well as the expanded action sequence.

- **coverage, env, transitive, infer** are all the same as in **ask** or **query**. Their default values are reasonable for the planner, and should only be changed if you know what you are doing.

When it succeeds, the planner returns a plan in the same format as the query it is passed, except that the complex tasks are all replaced with ground primitive tasks. So a plan has the form **(actionSequence (TheList <primitive action1> <primitive action 2>** …**))**. If it fails, the planner returns **:fail**.

If the **:return-reasons?** argument is true, the planner will also return a second value containing the hierarchical plan justifications for the plan steps. This is actually not the hierarchical plan, but the inverted plan that associates each plan step with its parent task. Although the planner can be configured to return the plan tree, the inverted plan turned out to be more useful in explaining and adapting plans.

As an example, the first Blocks World problem defined in the HTN planner flat files returns the following flat plan:

```
(actionSequence
 (TheList
  (doPickup Block2)
  (doStack Block2 Block3)
  (doPickup Block1)
  (doStack Block1 Block2)))
```

The 2nd returned value is the reasons for these steps:

```
(TheList
 (reasonForTask (doPickup Block2) (moveBlock (TheSet)))
 (reasonForTask (doStack Block2 Block3) (moveBlock (TheSet)))
 (reasonForTask (doPickup Block1)
                (reasonForTask (moveBlock (TheSet Block2))
                               (moveBlock (TheSet))))
 (reasonForTask (doStack Block1 Block2)
                (reasonForTask (moveBlock (TheSet Block2))
                               (moveBlock (TheSet)))))
```

There are several global variables that are useful for getting insights into how the planner works and for debugging plans:

- ***debug-htn*** causes status messages to be printed, if non-**nil**.
- ***break-on-backtrack*** will cause a breakpoint when the planner backtracks, if non-**nil**.
- ***trace-tasks*** will cause tasks that unify with members of ***trace-tasks*** to be traced, as will their subtasks.
- ***optimize-htn*** causes optimized Lisp code to be used for choosing the best expansion method, if non-**nil**. The default is **nil**, so that query is used to reason about what expansions are preferred.

For more details, please see the HTN planner documentation that comes with the FIRE installation.

5 Extending FIRE

FIRE is designed to be flexible. To that end, we have built in considerable support for extending its reasoning abilities. This section describes what you need to know in order to tinker "under the hood".

5.1 Adding a new reasoning source

The purpose of a reasoning source is to extend the capabilities of the reasoner, via a technique known as *procedural attachment*. That is, queries in essence invoke procedures, which do work that would be inconvenient, inefficient, or impossible to do entirely within the reasoner. When should one use a reasoning source?

- When interfacing an external system to FIRE. Examples include SME, sketching systems, geographic information systems, and web scrapers.
- When operations are heavily procedural. Just because reasoning systems can in principle do anything doesn't mean that one should make them do everything. An important engineering principle is to use the right tool for the right job.
- When there are substantial amounts of information that doesn't need to be in the KB. Large databases, for instance, don't need to be completely absorbed into a KB.
- When one is building a distributed reasoning system. Reasoning sources enable queries to access remote resources.

Reasoning sources should be organized around a set of related operations. For example, the analogy source, which is built into FIRE, provides an interface to SME and integrated implementations of MAC/FAC and SAGE. By using a reasoning source, specialized data structures (dgroups, smes, content vectors, case libraries, and retrievals) can be cached and incrementally reified on demand.

5.1.1 Deciding what predicates should be provided

The first decision is what predicates should be provided. This is an interesting tradeoff between two concerns: What the underlying system naturally provides, and what will lead to the most efficient queries. For example, suppose one is converting an old LTRE-based system to FIRE. LTRE rules often invoked procedures, via the **`:test`** keyword. Such procedures are good candidates for becoming predicates. Operations that cause the creation of data structures are another source of good candidates. Operations that retrieve information from complex data structures need to be considered with care, since their results will be reified in the reasoning system. Focusing exclusively on low-level accessors may lead to unnecessary cluttering of WM. What orders of computation should be supported becomes an important question. Some of the redundant predicates in the analogy ontology (see **`fire/flat-files/analogy-ontology.meld`**) exist to avoid excessive reification in queries, by compressing what would be several antecedents into one.

Predicates are defined in terms of procedures in the reasoning source. How many procedures one needs depends on what parameters will be provided as inputs versus produced as outputs. Suppose we want to define a unary predicate

```
(massNoun <x>)
```

which is true exactly when **<x>** is a mass noun. Logically, there are two cases we might handle here:

1. **<x>** is bound to something. The procedure has to test the argument, and signal true if it is indeed a mass noun.
2. **<x>** is a variable. The procedure returns a list of all the mass nouns.

The first one is clearly essential. The second one would make the system more logically complete, but also more likely to fail: There are a lot of mass nouns in a reasonable lexicon, and someone rarely means to list them all. It is better to be relatively incomplete and avoid costly failures than it is to be more complete. In general, if listing a large amount of information is going to be a necessary operation, it is wiser to provide a different predicate for doing that, so that it is very clear what one is doing.

Typically there are three patterns that procedures which implement predicates follow:

1. Test. The arguments are all bound, and the procedure tests to see if the property holds (if a unary predicate) or if the relationship holds (if multiple arguments).
2. Generate. All but one argument are bound, and the unbound variable is provided with value(s) that will satisfy the relationship.
3. List. Getting all of the alternatives. The results are bound as a list to the unbound variable.

Once you've decided what orders of computation to support, you're ready to begin implementing your reasoning source.

5.1.2 Implementing the predicates

As with many kinds of programming, careful study of an example yields rewards. We'll use examples from the analogy source to illustrate the principles involved.

5.1.2.1 Creating the interface

When an instance of a source gets created, the procedure which does this job has to call **register-ask-source**, which ties procedures to queries. There are two macros defined which simplify this process: **register-simple-handler** and **register-mc-handler**.

The only difference is that **register-mc-handler** takes both a mixed-case and hyphenated predicate[14]. For example, in macfac-accessors.lsp,

```
(register-simple-handler d::reminding
  run-macfac (:known :known :known :variable :variable) t)
```

binds the procedure **run-macfac** to the predicate **data::reminding**, assuming that the first three arguments (probe, case library, match constraints) are bound and the last two arguments (retrieved dgroup, matcher) are produced by the procedure. Similarly,

```
;; (caseLibraryContents ?library ?set)
(register-mc-handler
    d::caseLibraryContents d::case-library-contents
  ;; Accessor
  case-library-contents-find (:known :variable))
```

implements the ability to retrieve the contents of a case library, using **caseLibraryContents**. Once you've got the forms installed for each of the directions of computation for all of your predicates, it's time to write the procedures.

5.1.2.2 Creating the handler procedures

The procedures for implementing an order of computation are called *handler procedures*, since they handle part of the semantics of a predicate. They have a variable number of arguments – FIRE's internals will call them with

```
(<handler name> <rsource> <context> <number> <response> <effort>
                <query> . <other args>)
```

Where <*handler name*> is the procedure specified in the registration process, <*rsource*> is a pointer to the reasoning source instance itself (so that state internal to the source can be accessed, and the reasoner and KB can be accessed, although these are also bound by the usual default variables). <*response*> indicates the kind of response expected, <*effort*> is an effort specification, and <*query*> is the original form of the query. <*other args*> is the list of values that are passed into the query.

Since most of these are stereotyped, another macro is provided to simplify things: **defsource-handler**. All you have to provide are <*other args*>. Please see **sources.lsp** and examples in the FIRE source code for details.

Given that FIRE's working memory uses an LTMS, results from reasoning sources must be appropriately justified. If a result depends on other facts in the working memory, you should be sure to justify conclusions based on them. If a conclusion might change with subsequent incoming data, installing a closed-world assumption is a wise idea. Again, see

[14] Mixed-case predicate conventions are deprecated, and are unlikely to show up in future versions of FIRE.

the FIRE source code for details – for instance, **metaknowledge.lsp** and **planner\htn.lsp**.

5.1.2.3 Doing the ontological work

You also must define the structural properties of your new predicates, weaving them into the ontology appropriately. At minimum, you need to include the appropriate **isa** statements, a **comment** assertion, and **arity** plus argument constraints. It is also wise to provide **specPred** information, plus any other axioms that are needed to pin down the logical implications of your predicates. Should your predicates show up in analogical operations? If not, then it is important to add the appropriate analogy control predicates into an appropriate microtheory. These flat files need to be loaded once in the KB used by whatever system(s) are using your reasoning source. Since predicates are essentially global, it is important to keep to the conventions of the KB when choosing predicate names, and avoid conflicts.

5.2 Adding new quantifiers

FIRE's federated architecture means that it must carefully analyze query terms, so that the input and result signatures of predicates are respected. This means that its analysis routines must provide correct information about the predicates they encounter. One has to tell the system when adding a new quantifier, so that the local variable introduced is treated as a local variable, rather than as a free variable in the expression. One does this by extending the method **fire::introduces-local-variable?**, which returns t if a predicate introduces a local variable and nil otherwise. FIRE already defines this method over standard quantifiers, e.g.,

```
(defmethod introduces-local-variable? ((pred (eql 'data::forAll))) t)
(defmethod introduces-local-variable? ((pred (eql 'data::thereExists)))
t)
```

It is assumed that the quantifier will introduce a single local variable, and that local variable will be declared as the first argument in the expression. If you wish to introduce quantifiers that declare multiple local variables at once, or define them in some other location, you will have to modify the code in **formulas.lsp** more extensively.

6 Extras

6.1 Human-Readable Namestrings

Most applications eventually have to present information to users. The knowledge-bases we use have several facts assigning human-readable strings to concepts (**prettyString-Canonical**, **prettyString**, **prettyName**, **preferredNameString**, and **nameString** being the most common). Since fetching these namestrings occurs often in apps with user-interfaces, FIRE provides a few utilities for this purpose.

```
(get-namestring <source> <term> &key context)
```

get-namestring is the main public function for fetching user-readable namestrings for concepts. It returns two values:

1) The namestring associated with the given object. This will always return a string. If no namestring could be found, (namestring-fallback <source> <term> <context> will be called.
2) Non-nil iff a namestring was actually found for the obj. In other words, this will be nil if namestring-fallback had to be called.

The source parameter for get-namestring can be a reasoner or a KB. This is a method, so it can be specialized for other sources of information as well. For example, nuSketch defines get-namestring methods where the source is a sketch.

(get-namestring-from-source <source> <obj> <context>)

This method is used internally by get-namestring. You will probably never need to call this yourself, but it may be useful to specialize it for different sources or terms. Using nuSketch as an example again, because of its specializations, one make calls like (get-namestring <sketch> <glyph>) and it will correctly fetch the namestring associated with the given glyph.

(default-namestring-context <source> <obj> <context>)

Again, you will probably never call this method directly, but you can specialize it for various kinds of sources and objects.

(namestring-fallback <source> <obj> <context>)

This is called by get-namestring when get-namestring-from-source fails to find a namestring. Normally it just writes obj to a string, but you might find it useful to specialize it for other behaviors.

namestring-cache-mixin

This optional mixin class can be used with reasoners (and other sources of namestring information) to cache namestrings to speed up lookup. Fetching namestrings from the KB is slow enough that user-interfaces that present several namestrings at once will probably seem sluggish. This mixin caches the namestrings once they have been found.

To use, include it in your class definition and give it precedence over the reasoner, KB, or other namestring source. For example, the reasoner class used by nuSketch is defined as follows:

```
(defclass ns-reasoner (fire:namestring-cache-mixin
                       fire:reasoner)
  ((sketch
    :documentation "This slot is optional since not all
                    reasoners created within nuSketch are
                    for a specific sketch, but since every
                    sketch has its own reasoner, this slot
                    can be useful."
    :accessor sketch :initarg :sketch :initform nil)
    …))
```

Note that in the above definition, the namestring-cache-mixin is listed before reasoner in the parent classes of ns-reasoner. This ensures that the namestring-cache-mixin's get-namestring methods will be called instead of the normal ones used by the reasoner.

One downside of caching the namestrings is that the cache isn't smart enough to know when the namestring has changed. So you have to call the following whenever you change a namestring:

(reset-cached-namestring <source> <obj> <context>)

In nuSketch, for example, whenever a user renames a glyph (i.e. assigns a new namestring to it), nuSketch calls reset-cached-namestring.

7 Tips, Tricks, Traps, and Troubleshooting

7.1 Problems while getting set up

Problems in opening a new knowledge base: While the Plan B database is a lot more robust, it is still possible to trash it if you or your code work hard enough at it. In that case, you'll need to reinstall a new copy of the KB. It is important to delete the old KB first, since adding an arbitrary file from another KB to the database files in that directory will lead to undefined results. If you have been making substantial changes to a KB, it is advisable to back it up, by making a compressed archive periodically. There is also a journaling system that tracks changes to a KB, so it is possible to reinstall the KB you used as a starting point and reconstruct your changes with the journaling information. Generally this is done by systems not people (e.g., Companions internals), so please see the source code for more information.

Determining whether the KB is loaded into RAM or runs from disk: This was something that had to be set by hand with the old BDL database, but now AllegroCache is handling caching automatically. For some very intense KB changes, it can be worth fiddling with AllegroCache's parameters – see the KB build procedure (**`planb\kb-build.lsp`**) for an example. However, aside from KB building, we've not yet found places where changing the parameters from what we've set them to be by default makes a difference.

7.2 Problems found during shakedown

Problems are found with the evaluation subsystem: The most likely cause of such problems is that the assertions in the knowledge base that link Lisp handlers to specific evaluatable predicates are out of synch. The following procedure will handle these problems:

```
(fire::reinstall-fire-default-evaluation-info)
```

This procedure clears the old handler information out of the KB and reloads both the axioms and the code from the current source files. These source files are created automatically from the file **`evalfns.lsp`** in the **`fire\v3`** source directory. **`evalfns.lsp`** contains calls to a specialized macro that produces code and axioms for each evaluatable function. If you (or someone else) has updated **`evalfns.lsp`**, you need to call

```
(fire::create-fire-default-evaluation-files)
```

before calling the reinstallation procedure above, to ensure that the source code and axiom files are up to date.

Problems are found with analogical matching: Dehydrated copies of SME are stored in the tests subdirectory; comparing the current comparison against the latest known good one is an excellent way to figure out these problems. Use of **`rbrowse::browse-sme`** is strongly recommended.

7.3 Problems during development

You get a method not found error when calling fire:ask: Please make sure that you have opened a knowledge base and have created a reasoner. Having forgotten one or both of these steps is the primary cause of such errors.

There are missing predicates: You are probably missing a flat file. Axioms for FIRE KBs are loaded from ASCII text files. FIRE-based systems typically make a number of additions to the KB, since it is good programming style in this context to use declarative mechanisms as much as possible. Most FIRE-based systems have loaders that automatically load the correct set of flat files.

You've reloaded a file of rules and the old rules are still there: The model in FIRE is that adding a flat file does not remove old contents. In programming languages, loading a new definition of a procedure removes the old one. In logic, adding more axioms about a predicate only increases the specificity of what it means. If you want axioms removed, you have to remove them explicitly. This is yet another reason why it is good practice to segregate rules for a particular purpose into a distinct microtheory, so that the contents of that microtheory can be deleted and then reconstructed via loading updated flat files. See Section 4.4 for details.

You added knowledge to the KB, but it isn't there anymore after getting an updated KB: Periodically we rebuild the KB from scratch, using flat files that define the contents. If you didn't edit one of those files, or included your changes in a file that isn't part of the build process for that KB, it will not show up in the next version. If you have been making changes to the KB directly and journaling is turned on, you should be able to use the contents of the journal file to construct a flat file for more convenient reloading. However, we recommend that most development proceed by using flat files, checking predicate information using Rbrowse, to minimize errors. For example, while FIRE applications tend to not use the comments associated with the predicate, including comment statements is a crucial form of documentation for others reading through the KB or your flat files.

8 References

1. Anderson, W., Hendler, J., Evett, M. and Kettler, B. 1994. Massively parallel matching of knowledge structures. In Kitano, H. and Hendler, J. (Eds.) *Massively Parallel Artificial Intelligence*, MIT Press, pp. 52-72.
2. Everett, J. and Forbus, K. 1996. Scaling up logic-Based truth maintenance systems via fact garbage collection. Proceedings of the 13th National Conference on Artificial Intelligence.
3. Falkenhainer, B., Forbus, K., Gentner, D. ``The Structure-Mapping Engine: Algorithm and examples" *Artificial Intelligence*, **41**, 1989, pp. 1-63.

4. Falkenhainer, B., Forbus, K., and Gentner, D. The Structure-Mapping Engine. Proceedings of AAAI-86, Philadelphia, PA, August, 1986
5. Forbus, K. and de Kleer, J., *Building Problem Solvers*, MIT Press, 1993.
6. Forbus, K., Ferguson, R. and Gentner, D. (1994). Incremental Structure-Mapping. *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, August.
7. Forbus, K., Ferguson, R., Lovett, A., & Gentner, D. (in press). Extending SME to handle large-scale cognitive modeling. *Cognitive Science*.
8. Forbus, K., Gentner, D. and Law, K. 1995. MAC/FAC: A model of Similarity-based Retrieval. *Cognitive Science*, 19(2), April-June, pp 141-205.
9. Forbus, K., Gentner, D., Everett, J. and Wu, M. (1997). Towards a computational model of evaluating and using analogical inferences. *Proceedings of CogSci97*.
10. Forbus, K., Hinrichs, T., de Kleer, J., and Usher, J. (2010). FIRE: Infrastructure for Experience-based Systems with Common Sense. *AAAI Fall Symposium on Commonsense Knowledge*, Arlington, VA.
11. Forbus, K., Mostek, T. and Ferguson, R. (2002). An analogy ontology for integrating analogical processing and first-principles reasoning. *Proceedings of IAAI-02*, July.
12. Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science, 7*, 155-170.
13. Gentner, D. & Namy, L. L. (2006). Analogical processes in language learning. *Current Directions in Psychological Science*, 15(6), 297-301.
14. Kandaswamy, S., Forbus, K., and Gentner, D. (2014). Modeling Learning via Progressive Alignment using Interim Generalizations. *Proceedings of the Cognitive Science Society.*
15. Karp, P. and Paley, S. 1995. Knowledge Representation in the Large. *Proceedings of IJCAI-95*.
16. Kuehne, S., Forbus, K., Gentner, D. and Quinn, B. (2000). SEQL: Category learning as progressive abstraction using structure mapping. *Proceedings of CogSci 2000*, August.
17. McLure, M.D., Friedman S.E. and Forbus, K.D. (2015). Extending Analogical Generalization with Near-Misses. *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, Austin, Texas
18. Mostek, T., Forbus, K, and Meverden, C. (2000). Dynamic case creation and expansion for analogical reasoning. *Proceedings of AAAI-2000*. Austin, TX.

9 Appendix A: Vocabulary of specialized predicates in ASK and QUERY

Some predicates are "hard-wired" into **ASK** and **QUERY**. This appendix describes them.

9.1 Predicates handled specially by ASK

9.1.1 Structural knowledge from the KB

genls and **isa** statements are treated specially by ASK. These procedures use FIRE internals for efficiency. Special things to note for genls:

- **(genls <var> <var>)** will current grab every genls statement. This is almost always a mistake, and unless this is limited to a specific microtheory (see Section 4.5.1), FIRE treats it as a failure.
- **(genls <var> <constant>) and (genls <constant> <var>)** will only get immediate genls or specs. Use genlsTransitive or genlPredsTransitive if you really want the entire sublattice or superlattice.

Regarding **isa**,

- Like all variables input to genls, **(isa <var> <var>)** will seriously hurt you, unless this is limited to a single microtheory and lookup only. We will probably change this to return failure in the future.
- **(isa <constant> <var>)** will return the explicitly known collections for **<constant>**, rather than the deductive closure.
- **(isa <constant> <constant>)** does the appropriate chaining in the genls lattice, but does not invoke other axioms. More complex derivations of attributes/collection membership should be done through backchaining.

9.1.2 Structural predicates

Structural predicates are those which process the structure of assertions themselves. Most of this vocabulary has been drawn from the Cyc KB conventions, but we have introduced several new predicates to better support our work.

9.1.2.1 Equality and ordering on statements

(equals <a1> <a2> … <an>)
is true exactly when **<a1>** through **<an>** are all Lisp **EQUAL**.

(equalp <a1> <a2> … <an>)
is true exactly when **<a1>** through **<an>** are all Lisp **EQUALP**. (This does case-insensitive string comparisons, where equals does case-sensitive comparisons.)

(different <a1> <a2> … <an>)
is true exactly when none of **<a1>** through **<an>** are Lisp **EQUAL**.

(alphalessp <a1> <a2>)
is true exactly when **<a1>** is less than **<a2>**, in the ordering defined by the FIRE procedure **alphalessp**.

9.1.2.2 Accessing the structure of statements

Where predicates come from the Cyc KB, we have included their comments in the descriptions below, since they capture their designers' insights. Information about implementation is specific to FIRE.

(assertedTermSentences TERM SENTENCE)
means that **SENTENCE**, which is found in the KB, contains **TERM**.
Cases:

1. constant TERM, constant SENTENCE: Check to see that SENTENCE is in KB, and if so, does it contain term.
2. variable TERM, constant SENTENCE: Check if sentence is in KB, and extract all terms.
3. constant TERM, variable SENTENCE: Retrieve all references to TERM in KB, remove those where it doesn't appear as a term (i.e., when it is used only as a predicate).

(formulaArgument FORMULA N TERM)
means that **TERM** appears as the Nth argument in **FORMULA**.
Cases:

1. All constants: test if true.
2. Formula variable: punt, this is local.
3. N variable only: See if TERM is one of the args, and bind it appropriately.
4. TERM variable only: Bind it to Nth argument.
5. Both N, TERM variable: Generate solutions for each argument.

(justificationForStatementIn <statement> <expression> <antecedents>)
is true exactly when **<expression>** contains a justification for **<statement>**, indicating that **<statement>** should be believed as a consequence of the set of statements **<antecedents>**.

(natArgument NAT N TERM)
means that **TERM** is in the Nth argument position of the non-atomic term **NAT**. For example, **(natArgument (JuvenileFn Dog) 1 Dog)**. Note that **(termOfUnit NAT (FUNCTION ... ARGN ...))** implies **(natArgument NAT N ARGN)**. This predicate exists to make it easier and more efficient to write arity-independent rules about functional terms without having to resort to dotted variable syntax such as **(termOfUnit ?NAT (?FUNCTION . ?ARGS))**.

(natFunction NAT FUNCTION)
states that **FUNCTION** is the function used in the non-atomic term **NAT**. For example, **(natFunction (JuvenileFn Dog) JuvenileFn)**. More precisely, **(termOfUnit NAT (FUNCTION ...))** implies **(natFunction NAT FUNCTION)**. This predicate exists to make it easier and more efficient to write arity-independent rules about functional terms without having to resort to dotted variable syntax such as **(termOfUnit ?NAT (?FUNCTION . ?ARGS))**. Note: **natFunction** is like **operatorFormulas**, but its arguments are reversed.

(noArgumentHasPredicate ?fact ?pred)
is true iff there is no subexpression of an argument of **?fact** such that **?pred** is its functor. That is, it goes down the tree, not just one level. This is the negation of **someArgumentHasPredicate**.

(operatorFormulas TERM FORMULA)
means that **TERM** is the operator of **FORMULA**, i.e., it appears in the zeroth position. Cases:

1. constant term, constant formula: test
2. constant term, variable formula: Don't do anything. See **assertedTermSentences** instead.
3. variable term, constant formula: bind to operator of the formula.

(rationaleForOccurrenceIn <occurrence> <expression> <relation> <antecedents>)
is true exactly when the event or situation type **<occurrence>** is connected to enabling/disabling conditions **<antecedents>** by some subexpression of **<expression>. <relation>** is the relationship that holds between them, this is needed to determine the type of connection between the occurrence and the antecedents. **<relation>** will always be a **specPred** of **explains-Generic**. Currently **<relation>** and **<antecedents>** must be variables that will be bound via **ASK**.

(someArgumentHasPredicate ?fact ?pred)
is true iff there is some subexpression of an argument of **?fact** such that **?pred** is its functor. That is, it goes down the tree, not just one level

(subexpressionMatching <pattern> <expression> <subexpression>)
is true exactly when **<subexpression>**, which must unify with **<pattern>**, is a subexpression of **<expression>**.

(termFormulas SENTENCE TERM)
means that **TERM** appears somewhere in **SENTENCE**. This is a purely local query, hence it doesn't need to check to see if something is in the reasoner.

Cases:

1. Variable term, constant sentence: Generates assertions for each term in the sentence.
2. Constant term, sentence: True or false depending on whether term is in sentence.

The Cyc documentation is ambiguous about whether or not this is recursive. In FIRE we treat it as recursive.

(unifies <expression1> <expression2>)
is true exactly when its arguments unify.

(containsPattern <pattern> <expression>) is true exactly when some subexpression of **<expression>** unifies with **<pattern>**. The bindings of the pattern are available as part of the results.

(containsExpression *<subexpression>* *<expression>*) is true exactly when <*subexpression*> is a member of <*expression*>. This does not unify or bind variables.

(containsUnifyingArgument <expression> <pattern> <result-var>)
provides bindings for <*result-var*> such that they are arguments of <*expression*> or arguments of its subexpressions, recursively. Any variables appearing in <*pattern*> are also available as a result for each solution.

(individualRepresenting <defining contents> <string> <var>)
When used the first time, generates a unique symbol, prefixed by <string> and binding it to <var>. Subsequent queries within the same reasoner with the same defining contents will return the same symbol. This provides the equivalent to the "find or make" functionality when introducing new atomic terms.

9.1.3 Metaknowledge predicates

Metaknowledge predicates provide information about the state of knowledge in the system. The metaknowledge predicates implemented in ASK are:

(knownSentence *<sentence>*)
is true exactly when ***<sentence>*** can be found in the WM or KB through lookup alone.

(unknownSentence *<sentence>*)
is true exactly when ***<sentence>*** cannot be found in the WM or KB.

(uninferredSentence *<sentence>*)
is true exactly when **<sentence>** cannot be derived by current means. Open variables in <***sentence***> are not allowed.

(trueSentence *<sentence>*)
is true exactly when <sentence> can be derived.

(falseSentence *<sentence>*)
is true exactly when ***<sentence>*** is either known to be false or cannot be derived by current means.

(consistentThat *<sentence>*)
is true exactly when ***<sentence>*** is not already known to be false.

One use of these predicates is handling negation by failure, which is an approximation to true negation, as per **uninferredSentence** and **unknownSentence** above.

9.1.4 The Eval subsystem

Sometimes it is wise to render unto procedures what is procedural. Arithmetic operations on numbers, list operations, and closed-world assumptions are all good examples. FIRE's Eval subsystem supports this via ASK through a specialized predicate **evaluate** which takes two arguments, a value and an expression to be evaluated. When the value is a variable, the result of evaluating the expression is bound to that result. When the value isn't a variable, the result of evaluating the expression is compared to the value, and if they are the same the evaluate statement is justified as true[15].

The functions and relations handled by the Eval subsystem are those which in Cyc are considered evaluatable-functions. The subset we have implemented includes the following:

- *Procedures on sets and lists:* **LengthOfListFn**, **CardinalityFn**, **ListFn**, **MemberFn**, **SublistFromToFn**, **NthInListFn**, **PositionInListFn**, **RestOfListFn**, **TheList**, **TheSet, JoinListsFn**, **ReverseListFn**, **SortFn**, **SetOrCollectionUnion**, **SetOrCollectionIntersection**, **MapFunctionOverList**, **FormulaArgListFn**, **MakeFormulaFn**.
- *Procedures on numbers:* **PlusFn**, **TimesFn**, **DifferenceFn**, **QuotientFn**, **AbsoluteValueFn**, **ExponentFn**, **ExpFn**, **LogFn**, **MaximumFn**, **PlusAll**, **Average**
- *Higher-level procedures:* **FunctionToArg**
- *Non-monotonic predicates:* **TheClosedRetrievalSetOf**. (Not in Cyc. Value denotes the construal of set, based on statements explicitly known when the evaluation occurs. A timestamped CWA is provided for later reasoning about whether or not the construal should be recomputed.

This list is not exhaustive, the current set is in **evalfns.lsp**. This file is processed by FIRE internal procedures to construct two files, **evaluate-handlers** and **evaluate-axioms**, using the procedure **(fire::create-fire-default-evaluation-files)**. Calling the procedure **(fire::reinstall-fire-**

[15] If the values do not match, via the Lisp EQUAL predicate, the ASK fails. We do not install the failed evaluation in the working memory as false.

default-evaluation-info) causes the procedure definitions in evaluate-handlers to be loaded, and for the axioms in **evaluate-axioms** to be installed in the kb.

If you want to quickly try out an expression to see what it will do, the lisp procedure **fire-evaluate** takes an expression (i.e. the second argument to evaluate) and an optional microtheory, and returns the value that the evaluate subsystem computes for that expression.

9.1.5 Dynamic Update Predicates

Dynamic Update predicates always compute their results and never look for results in working memory. Dynamic update predicates implemented directly in ask are:

(currentUniversalTime <time>)
binds **<time>** to the lisp-encoding of the current universal time. Because this is a lisp-specific opaque representation, storing such values in the kb is discouraged.

(wmAntecedentOf <consequent> <antecedent>)
checks working memory justifications to either verify that **<antecedent>** is a justification for **<consequent>**, bind the antecedents of **<consequent>**, or bind the consequences of **<antecedent>**. At least one of the arguments must be bound, and all bound arguments must be believed true.

9.1.6 Binding List Predicates

Binding lists inside the FIRE and Plan B implementations are implemented as alists, and reified via terms constructed using **TheSet** and **TheList**. Reified binding lists are typically produced via outsourced predicates, given the use of a truth maintenance system in working memory, implementing elementary list operations via Horn clauses is not recommended (see the **eval** subsystem above). The following predicates are useful for testing binding lists:

- **(sameBindings *<b11> <b12>*)** holds when ***<b11>*** and ***<b12>*** are both the same binding list.
- **(differentBindings *<b11> <b12>*)** holds when ***<b11>*** and ***<b12>*** differ in some binding (or the presence of a binding).
- **(subsetOfBindings *<b11> <b12>*)** holds when ***<b11>*** is a subset of the bindings found in ***<b12>***.

9.2 Predicates handled specially by QUERY

Like **ASK**, **QUERY** handles conjunctions expressed via **and**.

9.2.1 N-ary predication

There are times when one wants the reasoning equivalent of EVERY and SOME in Lisp. For instance, evaluating when an action is possible requires gathering the set of its preconditions and establishing whether or not they hold. These predicates provide this service.

(everySatisfies <var> <list or set> <statement>)
(someSatisfies <var> <list or set> <statement>)
everySatisfies holds if for every element of **<list or set>,** substituting it for **<var>** in **<statement>** is provable.
someSatisfies holds if this is true for any element of **<list or set>**.

Example:

```
cl-user(93): (fetch-trues '(foo ?x))
((foo b))
cl-user(94): (fire:q '(someSatisfies ?x (TheSet a c) (foo ?x)))
nil
cl-user(95): (fire:q '(someSatisfies ?x (TheSet a b) (foo ?x)))
((nil (someSatisfies ?x (TheSet a b) (foo ?x)) :step (:WM (foo b))))
cl-user(96): (why? '(someSatisfies ?x (TheSet a b) (foo ?x)))

(someSatisfies ?x (TheSet a b) (foo ?x)) is true via bc on
   (foo b) is true
<N-3>
cl-user(97):
```

Current Limitations:

1. It will bail if there are any free variables in ***<statement>*** besides ***<var>***. I didn't want to deal with multiple solutions.
2. Negations still aren't handled.
3. It is agnostic with regard to **TheList** or **TheSet** for ***<list or set>***, but the functor of that term must be one of them.

9.2.2 Metaknowledge predicates handled by QUERY

(forEffectOnly <sentence>)

f**orEffectOnly** is used to run a query prior to collecting elements in a closed retrieval set, or for establishing preconditions when using **solve**.

10 Index

break-on-backtrack 43
debug-htn 43
default-suggestions-mt 38, 40
kb 13
optimize-htn 43
reasoner 13
record-retrievals 31
results 14, 36
solve-max-agenda-work 38
solve-max-depth 38
trace-tasks 43
<== 36
AbsoluteValueFn 57
`actionSequence` 42, 43
add-source 15
AllAssumptions 24
AllegroCache parameters 50
allFactsAllowed 23
alphalessp 54
Analogical matching 25
Analogical retrieval 30
analogy ontology 8
AnalogySkolemFn 27
arg-isa 20
arity 20
ask 13
  controlling effort 22
ask-it 13, 21
assertedTermSentences 54
assumptions-of 16
atomicAnalogyNat 26
Average 57
BaseKB 10
baseOfMatch 27
bestMapping 27
Blocks World 43
browse-current-sme 14
browse-kb 14
candidateInferenceContent 27
candidateInferenceExtrapolationScore 27
candidateInferenceOf 27
candidateInferenceSupportScore 27
CardinalityFn 57
Case libraries 31
caseLibraryContains 31
CaseLibraryMinusFn 31
CaseLibrarySansFn 31
CaseLibraryUnionFn 31
close-kb 13, 17
collections-of 15
ComplexActionPredicate 42
consistentThat 57
containsPattern 56
contextEnvAllowed 23
correspondsInMapping 27
cost evaluation 41
create-fire-default-evaluation-files 57
currentUniversalTime 58
debugging
  analogical matching 50
  ask 51
  corrupt KB 50
  evaluation subsystem 50
  missing knowledge 51
  missing predicates 51
  old rules/knowledge persisting 51
  overloading constants 19
  plans 43
  too many open variables 37
default-estimate-ao-node-cost 41
default-namestring-context 48
defSuggestion 39
DifferenceFn 57
different 54
dynamic update predicates 58
entitySupportedInferencesAllowed 24
entitySupportedInferencesNotAllowed 24
equals 53
estimate-ao-node-cost 41
evaluate 57
evaluate-axioms 57

evaluate-handlers 57
evaluating expressions 57
everySatisfies 59
EverythingPSC 10
exactMatchOnly 23
excludedCorrespondence 25
excludedCross-PartitionCorrespondences 26
ExpFn 57
explain-fact 14
ExponentFn 57
falseSentence 57
fetch 16
fetch-trues 16
fire-evaluate 58
Flat files 18
forEffectOnly 59
forget-fact 18
forget-mt 18
FormulaArgListFn 57
formulaArgument 54
FunctionToArg 57
genls 53
get-namestring 47
get-namestring-from-source 48
get-solution 38
groundOnly 23
handler procedures 46
honorTimestamps 24
human-readable strings 47
identicalFunctions 25
ignoreTimestamps 24
individualRepresenting 56
inferenceAllowed 23
informant-of 16
in-kb 16
inKB 6
in-microtheory 19
in-reasoner 15
instance-of? 16, 20
introduces-local-variable? 47
isa 53
ist-Information 10
JoinListsFn 57
justificationForStatementIn 54
Karla the Hawk 25
kbOnly 23
kb-store 17
knowledge-level programming 19
knownSentence 56
LengthOfListFn 57
ListFn 57
localOnly 23
LogFn 57
lookupOnly 23
MAC/FAC 30
make-fire-kb 12, 16
MakeFormulaFn 57
make-reasoner ..... *See*
MapFunctionOverList 57
MappingFn 26
matchBetween 25
MatcherFn 26
MaximumFn 57
meld-file->kb 18
MemberFn 57
microtheories 7
MinimalAssumptions 24
namestring-fallback 48
namestrings 47
natArgument 54
natFunction 55
negation 36
negation by failure 57
next-solve-step 38
n-instances-of 20
n-instances-of-transitive 21
noArgumentHasPredicate 55
NoAssumptions 24
noMinimalAscension 24, 28
nonTransitiveInference 23
notForAnalogy 26
NothingPSC 10
n-statements-of 21
n-statements-of-transitive 21
NthInListFn 57
nuke-kb-item 18
numAnswers 24
numberOfCorrespondences 27
ontology
  extending 47

open-kb 16
operatorFormulas 55
outsourced predicates 6
outsourcedOnly 23
Overloading constants 19
packages 19
pause-ao-tree 39
plan 42
PlusAll 57
PlusFn 57
PositionInListFn 57
procedural attachment 44
Prolog
  comparison with 36
q 36
QP theory 41
qrgsetup.lsp 10
quantifiers 47
query
  details 35
QuotientFn 57
rationaleForOccurrenceIn 55
Rbrowse 11
reasoning sourc 44
register-ask-source 15, 45
register-tell-source 15
reinstall-fire-default-evaluation-info ... 58
reminding 30
requiredCorrespondence 25
requireWithinPartitionCorrespondences 26
Resource limits
  backchaining depth bound 35
RestOfListFn 57
result-isa 20
retrieve-all 17
retrieve-it 17
retrieve-references 17
reverseCandidateInferenceOf 27
reverseCIsAllowed 24
ReverseListFn 57
run-solver-n-steps 39
run-to-solution 39
SAGE 32
sageAddExemplar 33
sageAddExemplarToContext ... 33
sageEntryPattern 34
sageExemplars 33
sageGeneralizations 33
sageStrategy 34
sageThreshold 34
sageUseProbability 34
SetOrCollectionIntersection 57
SetOrCollectionUnion 57
setup-solve-aotree 38
shakedown-fire 12
SME 8
solar system/Rutherford atom analogy 25
solve 37, 38
  debugging 42
solveAll 40
solveSequentially 41
someArgumentHasPredicate 55
someSatisfies 59
SortFn 57
spindle microtheory 19
Structural queries 20
structuralEvaluationScoreOf 27
subexpressionMatching 55
SublistFromToFn 57
subset-of? 20
suggestionGoalForm 40
suggestionResultStep 40
suggestionSubgoals 40
targetOfMatch 27
termFormulas 55
termOfUnit 54
TheClosedRetrievalSetOf 57
TheList 57
TheSet 57
TimesFn 57
trueSentence 56
ubiquitousForAnalogy 26
unifies 56
uninferredSentence 56
UniversalVocabularyMt 10
unknownSentence 56
unpause-ao-tree 39
useMinimalAscension 24, 28

useTransitiveInference 23
why? 16
withAbduction 24
withAbductivePolicy 24
withAbductivePredicates 24
withBackchainingDepth 24
withCounterfactual 24
with-kb 16
withMinimalAscensionMultiplier 28
with-reasoner ..... *See*
withTimeout 24
wmAntecedentOf 58
wmOnly 23
Zgraph 11