Appendix E: Implementation

This appendix discusses some of the issues of implementing the constraint-understanding system. It is not a complete description of the implemented system, and does not concern itself with algorithmic details, the most important of which are described in Appendix D. It just highlights some of the decisions that were taken regarding the languages and tools that were used to implement the system, issues that the other chapters and appendices in this thesis have abstracted away from.

Section E.1 justifies the decision not to use a standard CBR shell to implement the constraint-understanding system. Section E.2 justifies the choice of Java[1] as the particular programming language in which the system was implemented, and briefly describes the program. Section E.3 discusses the use of ExJava to parse EXPRESS [34] models, Section E.4 discusses the use of the Java Compiler Compiler (JavaCC)[2] to build an FIL parser, and Section E.5 discusses the generation of frame browsers in HTML.

E.1 Not Choosing a Standard CBR Shell

An alternative way of implementing the constraint-understanding system which was rejected is to use a standard CBR (case-based reasoning) shell, rather than writing a CBR engine from scratch. There are several such programs in existence, and they are surveyed in Chapter 6 of Watson [84].

For instance, ReMind (Section 6.10 of Watson [84] and pages 613/614 of Kolodner [39]) provides a library of CBR facilities in the form of C functions, so that ReMind can be used as a substratum for CBR programs. It represents cases as attribute-value pairs, and provides a variety of indexing, retrieval and adaptation facilities.

1. "Java" is a trademark of Sun Microsystems, Inc.
2. "Java Compiler Compiler" and "JavaCC" are trademarks of Sun Microsystems, Inc.

The main problem with using a standard CBR shell is that understanding the constraints on EXPRESS models is a particularly unusual CBR problem, as explained in Section 4.2 and throughout Appendix D. Retrieval is based on an unorthodox shallow indexing scheme (see Appendix D, Section D.1.3), matching operates on a pass-or-fail basis rather than computing a degree of match (see Appendix D, Section D.2), and cases can contain variables, which can be unified (see Section 6.3). These unorthodoxies render standard CBR shells unsuitable, and it would be very difficult and inappropriate to attempt to customise them to this application.

Using a standard CBR shell was also rejected due to lack of flexibility. Using a standard CBR shell would tie the constraint-understanding system to a particular programming language (although it may be possible to obviate this problem by using a glue language such as Perl (see Section E.2.1.2)). Worse still, it would tie the constraint-understanding system to a particular case representation formalism, not allowing the use of EXPRESS and FIL for representing cases (it might be possible to convert between the CBR tool's language and EXPRESS/FIL at runtime, much as ExJava objects are converted to frames (see Section E.3) in the current system, but that would be an additional programming burden and would detract from the performance of the constraint-understanding system).

So, instead of using a standard CBR shell, the constraint-understanding system was programmed from scratch, using Java. This is discussed in the next section.

E.2 Implementing the Constraint-Understanding System in Java

This section discusses the implementation of the constraint-understanding system in Java. Section E.2.1 justifies the choice of the object-oriented programming paradigm and the choice of Java in particular. The Java implementation of the constraint-understanding system is then briefly described in Section E.2.2.

E.2.1 The Choice of Programming Paradigm and Language

It was decided to implement the constraint-understanding system in Java (Grand [30]), an object-oriented programming language. This subsection justifies this decision, discussing some of the alternatives that were rejected, in terms of both programming paradigms and particular languages.

E.2.1.1 Choosing Object-Oriented Programming as the Paradigm

In object-oriented languages, a program is defined as a collection of objects, which encapsulate state, and provide methods which are abstract operations on those objects (Section 9.2 of Louden [44]). Objects and their methods can be understood independently of the way they are represented as concrete data structures and implemented as algorithms - these ideas are known as *data abstraction* and *procedural abstraction* respectively (Section 10.2.1 of van Vliet [79]). Object-oriented programs are thus decomposed primarily with respect to objects (within objects, methods may be functionally decomposed, which is secondary). Procedural programs, on the other hand, are decomposed primarily with respect to function, and lack data abstraction.

Using objects rather than functions as the primary unit of decomposition:

- is a more natural way of modelling the concepts in a domain;
- helps to smooth the transition between specification, design and implementation;
- allows the concrete representation of data to be hidden (*encapsulated*) within objects (Section 10.3.4 of van Vliet [79]).

Encapsulation makes objects easier to comprehend than naked data structures which are manipulated directly by the procedures that use them. Because the way objects are represented as concrete data and managed by methods is separated from the way they are used by client objects, objects are also more reusable and more adaptable to change than naked data structures.

Although object-oriented programming is not the only paradigm which supports encapsulation (for instance SML, a functional language, provides data abstraction through abstract data types (Section 6.5 of Myers, Clack and Poon [50])), its advantages do not end there. Object-oriented programming also provides *inheritance* (Section 2.2.2 of Eliëns [20]), which allows one class of objects to be declared as a subclass of another. A subclass inherits its state variables and methods from its superclass. Subtyping arises naturally in real-world domains, and inheritance further promotes reuse by avoiding having to duplicate code for subclasses of a common superclass. Instead of duplicating code, subclasses inherit from superclasses, and can add further state variables and methods, or override existing ones as necessary.

E.2.1.2 Choosing Java as the Language

In order to reap the benefits of object-oriented programming discussed above for the constraint-understanding system, an object-oriented programming language was the obvious choice. This is not as facile as it sounds, because it is one thing for a language to *enable* object-oriented programming, and another for it to *support* the paradigm (Section 1.2 of Stroustrup [72]). Even procedural programming languages such as C (Charlton, Leng and Little [13]) *enable* object-oriented programming, because it *is* possible to program in an object-oriented style in them. However, such languages do not particularly encourage object-oriented programming because they do not provide constructs to *support* it.

Some languages support the object-oriented paradigm more than others. C++ (Stroustrup [72]), a strong contender for the implementation language, is essentially C extended with object-oriented constructs to support encapsulation and inheritance. This encourages object-oriented programming but does not enforce it, as the programmer can easily revert to procedural programming using just the C subset of the language. Java, on the other hand, imposes to a greater extent an object-oriented style, because it is not possible to have global variables or functions which are not members of a class. It also has other advantages over

C++, such as:

- *Interfaces*, which specify what methods a class which implements that interface must define. Although a class can only have one direct superclass (unlike in C++, which allows multiple inheritance), it may implement any number of interfaces. Interfaces thus provide the benefits of multiple inheritance without the drawbacks (see Section 5.3.6 of Grand [30]).
- Java has *references* instead of pointers, which means that the programmer never has to be concerned with actual numeric addresses in the computer's memory. This eliminates bugs that can arise in languages such as C and C++ which allow pointer arithmetic. Unlike C++, Java does not distinguish between an object and a pointer to an object (all object variables in Java contain *references* to objects[3]), which is less confusing to humans.
- Garbage collection in Java is automatic (Java uses on-the-fly mark-sweep garbage collection), which gives the programmer less to worry about than in C and C++, for example. In C and C++, the programmer has to deallocate memory explicitly through pointers, which is a source of treacherous bugs (difficult to trace because they can manifest themselves much later than the culpable program statement).
- Java supports *reflection* by enabling the class of an object to be examined, and through the JavaBeans[4] architecture (Englander [22]). Java does have the disadvantage of not having templates like in C++, but atones for this by having a class `Object` which subsumes all classes. This facilitates writing generic algorithms such as the matching algorithm described in Appendix D, especially as it is possible in Java to test the type of a given object at runtime. Java provides very good support for introspection compared to most languages. However, the Java reflection mechanism is not flexible enough to avoid the need to provide the frame system (Chapter 5).

---

3. Or null.
4. "JavaBeans" is a trademark of Sun Microsystems, Inc.

Java compilers are unorthodox in that they do not convert the source code to machine code, but rather to a machine-independent bytecode format. This is a compromise between compilation and interpretation: the relatively minor disadvantages of needing a Java bytecode interpreter and the inevitable loss of execution speed (as compared to stand-alone machine code), against the advantage of extreme portability, which in turn promotes reuse.

Another alternative which was considered as the implementation language for the constraint-understanding system was Perl (Wall, Christiansen and Schwartz [81]). Perl is designed to "make the easy jobs easy, without making the hard jobs impossible" (Preface of Wall, Christiansen and Schwartz [81]). The main advantages of Perl are:

- Perl is particularly useful for text processing, as the language supports the matching of regular expressions. This could be useful for parsing and case matching. It does not, of course, add anything to what could be achieved in principle in Java, but the fact that regular expressions are supported by the language is conducive to rapid prototyping.
- Perl is interpreted rather than compiled (except for a quick compilation into an intermediate format at the start of each run of the program under the interpreter). Not having to wait through a lengthy compilation every time the program is changed certainly shortens program development time.
- Perl is a 'glue' language: it supports integration with other programs (whatever language they were written in) by allowing operating system shell commands to be embedded in Perl code. Different parts of a system could be implemented in different languages as appropriate, e.g. the constraint-understanding system could be implemented largely in Perl, but use Prolog (Sterling and Shapiro [71]) for unification.

Perl is a powerful language in terms of what can be done easily with it (not in terms of speed of execution), but it is not a very elegant language because:

- It is weakly typed - every variable is simply constrained to be a scalar, array, hash, subroutine or typeglob, as specified by prefixing the variable name with a special character (`$`, `@`, `%`, `&` or `*`).

- It is notoriously easy to write obfuscated code (especially in regular expressions).
- An occurrence of an array variable can mean either the array itself or the number of elements in the array, depending on whether it is evaluated in an array context or a scalar context.
- The rules of precedence are confusing and counterintuitive (e.g. "`rand 10 * 20`" means "`rand (10 * 20)`", whereas "`rand(10) * 20`" means "`(rand 10) * 20`" (page 78 of Wall, Christiansen and Schwartz [81])).
- Subroutines do not specify their formal parameters and return types. Instead, the actual parameters are passed by reference, and are visible within the subroutine in a special array variable (`@_`).
- Perl does not support record structures. The programmer has to set up hashes instead, using hash keys as field names. While this is more flexible than, for example, structures in C and C++, it is better to have rigid structures (i.e. with records and their fields hardwired into the program using special constructs, which are more readable) in the majority of circumstances where such flexibility is not needed.
- Perl does provide some support for object-oriented programming, using packages as classes and subroutines as methods, but the syntax is ugly and an object-oriented style is not encouraged, but rather provided as a cheap add-on.

Java was chosen over Perl primarily for its greater elegance and better support for object-oriented programming. The Author believed that this would pay more dividends in the long term than the 'quick and dirty' solution of using Perl as the implementation language.

Using Java also has more pragmatic advantages, such as the fact that it is currently the most heavily-used language in the Author's research group, with many tools available to support it. The constraint-understanding system is able to use an existing EXPRESS parser written in Java, ExJava (as discussed in Section E.3). The popularity and availability of Java also makes it more likely that the constraint-understanding system will in turn be used by others in the future.

A major advantage of Java is that it has libraries of standard modules (as do C++ and Perl) which are available for use in new programs. This promotes reuse because it encourages the programmer not to reinvent the wheel, as it were. For example, the constraint-understanding system uses objects such as hash tables, vectors, enumerations and input/output facilities from the standard Java packages.

Having discussed the choice of Java in this subsection, the implementation of the constraint-understanding system as a Java program will briefly be described in the next subsection.

E.2.2 The Constraint-Understanding Java Program

This subsection gives an overview of the structure of the constraint-understanding system as implemented as a Java program. It briefly describes the important classes in the program, and how they interact with one another.

As mentioned in Section 4.2, there are two entry points to the constraint-understanding system. There are two corresponding Java classes with `main` functions:

- `Constraints`: This is the usual entry point to the constraint-understanding system, where the user provides as input an EXPRESS model on which the constraints are to be understood.
- `FIL`: This entry point is used to add cases to the case library (these are *seed cases* - i.e. written by hand, in FIL). It is called `FIL` because this class implements the FIL parser (Section E.4).

These two classes consist solely of static[5] variables and static methods - only one instance of each of these classes is created each time the program is run. The `Constraints` class is responsible for top-level control of extraction (see Appendix D, Section D.1).

5. ‘Static’ is Java-speak for class variables (which have a single value, shared by all instances of a class) and class methods (which can be called independently of any instance of a class).

The case library is stored in a binary file which holds the cases in the system (see Appendix C) and also, for convenience, the parsed frame class models (see Appendix A). This binary file is an external format for the internal frames in the system, created using Java's serialization [sic] mechanism (Chapter 5 of Englander [22]). When invoked at the `Constraints` entry point, the system reads the binary file; the binary file is created and modified when the system is invoked at the `FIL` entry point. The use of a binary file saves having to parse the cases and frame class models from text files every time the system is invoked at the `Constraints` entry point (to process an EXPRESS model).

When invoked at the `Constraints` entry point, the system goes through five major phases (each of which can take a considerable length of time for large EXPRESS models, so the system reports the boundaries of these phases to the user):

- Firstly, the system loads the case library binary file.
- Secondly, the system parses the input EXPRESS model.[6] As explained in Section E.3, this is done using an pre-existing EXPRESS parser, ExJava, which results in a network of objects representing the EXPRESS model.
- Thirdly, the system converts the ExJava objects to frames representing the EXPRESS model (the conversion process will be discussed in Section E.3; the EXPRESS frames were described in Section 6.1).
- Fourthly, the system navigates the EXPRESS model frame network, extracting higher-level constraints (HLCs), as described algorithmically in Appendix D.
- Finally, the system outputs all the frames in memory as HTML files (as described in Section E.5). These frames include the augmented model (i.e. the frames representing the input EXPRESS model, and the HLC frames extracted from the model).[7] A system built on the constraint-understanding system which used the augmented model would proceed from here, in lieu of this step.

6. Parsing the EXPRESS model is independent of loading the case library, so parsing can be done first if the user prefers.
7. These frames also include the cases in the case library and the frames representing the frame class models of Appendix A, which will have been loaded in from the case library binary file.

The most important class in the constraint-understanding system is `Case`, whose instances represent cases in a case library. A `Case` object holds a frame of frame class `case_frame` to represent the case,[8] as described in Section 6.3. It has methods to match a prompt frame to the 'problem' part of a case (see Appendix D, Section D.2), and to extract the HLC suggested by the 'solution' part if the match succeeded (see Appendix D, Section D.3).

There is also a class `HashtableStack` to encapsulate the stack of hash tables used for the matcher's retraction mechanism for frame associations and variable bindings (see Appendix D, Section D.2). `HashtableStack` provides methods to push a new hash table onto the stack, to pop the top hash table off the stack, to store a frame association or variable binding in the top hash table, and to look up an associated frame or a variable's binding in each hash table until it is found.

The class `CaseLibrary` encapsulates the case library, where the `Case` objects are stored. It hides the details of how the case library is organised (changing the organisation would require only the `CaseLibrary` class to be edited), and how retrieval and insertion of cases are implemented. The methods of `CaseLibrary` are:

- The `retrieve` method takes a prompt frame and returns a vector of cases that could potentially match the prompt (the vector returned is required to be a superset of those cases that actually do match);
- The `insert` method takes a `Case` and adds it to the case library;
- `CaseLibrary` is also responsible for saving the case library (and the frame class models) in a binary file, and for loading that file back into the computer's memory.

The most important classes of the underlying frame system are:

- `Frame`, which encapsulates a frame. A `Frame` object records the frame's frame class (an instance of `FrameClass` below), its slot fillers (stored in a hash table), its internal instance number (see Section 5.3), and some other book-keeping information which is

8. This is the instance variable `_caseFrame` referred to in Appendix D.

not worth describing here. It provides methods to return the filler of a given slot and to set a given slot to a given value, and also to write the frame out as an FIL instance.

- `FrameClass`, which represents a frame class. A `FrameClass` object holds the frame which represents the frame class (a 'meta' frame which results from parsing an entity in the frame class EXPRESS model, as described in Section E.3). The main purpose of a `FrameClass` object is just to exist for its instance frames, but it also provides methods to test whether the frame class subsumes (i.e. is the same as, or a superclass of) another, to inquire what slots an instance frame is supposed to fill, to look up a `Slot` object for this frame class by name, to look up a `FrameClass` by name (a static method), and to return the frame representing this frame class (which would be useful for introspective reasoning).
- `Slot`, which represents a slot in a frame class. A `Slot` object holds the frame which represents the slot (a 'meta' frame which results from parsing an attribute in the frame class EXPRESS model, as described in Section E.3).

The not-so-important classes in the frame system are for representing special values such as defined type instances in FIL (see Section 5.3.3), enumeration symbols, variables in cases, and null values which are explicitly specified in cases (which are encoded internally as objects of class `Null`) as opposed to being omitted (encoded internally as null references). There are also various exceptions (objects that get thrown when something goes wrong, and have to be caught later so that they can be dealt with accordingly). More classes will be revealed in the next section, which discusses the use of ExJava to parse EXPRESS models.

The source code and documentation generated by Javadoc [73] for all classes are available for browsing on the Author's web site (Broad [8]).

E.3 Using ExJava to Parse EXPRESS Models

The constraint-understanding system reuses an existing tool written in Java to parse EXPRESS models. ExJava (Haslam [32]) was written in the Author's research group, and the compiled bytecode files are available within the group to be imported into Java programs that use ExJava.

ExJava parses an EXPRESS model, creating a network of Java objects to represent the syntactic constructs of EXPRESS (effectively an abstract syntax tree (Section 4.3 of Louden [44])), which a program that uses ExJava can navigate. However, what the constraint-understanding system needs is a network of frames rather than raw Java objects - a representation which is flexible enough for the generic reasoning processes of the constraint-understanding system (see Section 5.1). So the constraint-understanding system first uses ExJava to parse an EXPRESS model, and then converts the ExJava objects to frames - a very mundane process which is too tedious for this thesis to describe in detail. The reader only needs to understand that an EXPRESS model is read in, and ends up being represented internally as frames conforming to the EXPRESS metamodel in Section A.1 of Appendix A.

The frame system has a subclass of `Frame` for every EXPRESS construct that it needs to represent. The mapping between ExJava objects (e.g. of class `Entity`) and EXPRESS frames (e.g. of frame class `entity_type`), and between EXPRESS frames and objects of the corresponding subclass of `Frame` (e.g. of class `EntityFrame`) is more or less one-to-one.

It should be noted that there is not a subclass of `Frame` for every frame class in the frame class model. Frame classes only need to be declared to the system in EXPRESS files at runtime, not hardwired into the program. Only the EXPRESS frame classes (Appendix A, Section A.1) need to have subclasses of `Frame`; these subclasses exist solely for the conversion of ExJava objects to frames.

Each subclass of `Frame` provides a method to build a frame for the corresponding type of ExJava object. A global value-conversion function[9] is called as the static conversion method of the `Constraints` class iterates over the ExJava objects produced by parsing the EXPRESS model. This value-conversion function does a case analysis of the type of object that it is called with, constructing an instance of the appropriate subclass of `Frame`.

This section has discussed the use of ExJava as an EXPRESS parser. The next section discusses the use of JavaCC to generate a parser for FIL.

E.4 Using JavaCC to Generate the FIL Parser

The constraint-understanding system needs a parser for FIL (see Section 5.3). There is a compiler-compiler, JavaCC [74] which takes as input a specification of a textual language, and produces as output a Java program to parse that language.[10]

A JavaCC program is essentially just a grammar in which each production is tagged with an action to execute if that production is recognised in the input. JavaCC even allows actions in the middle of the right-hand side of a production, to be executed when the production has been partially recognised up to that point (JavaCC generates left-to-right parsers, with a default lookahead of one token which can be overridden for individual productions if necessary).

The use of JavaCC to produce a parser for FIL is very straightforward. The JavaCC program is essentially just the grammar from Figure 5-12, tagged with actions to construct frames when FIL instances are recognised and fill their slots. The actions automatically

9. Actually a static method of the class `Values`, which is not meant to be instantiated.
10. This is, in fact, an example of code generation (c.f. Section 2.1).

create a `Case` object for frames of frame class `case_frame` and insert them into the case library. The JavaCC source code for the FIL parser is included as Section E.4.1. (It defines the public class `FIL` referred to in Section E.2.2.)

E.4.1 JavaCC Source Code for the FIL Parser

```
     ///////////////////////////////////////
    // THE CONSTRAINT-UNDERSTANDING SYSTEM //
   // Written by Andrew Broad           ///
  //                                   ////
 // FIL.jj (1998:12:02 11:45)         /////
//////////////////////////////////////////////
// Methods of FIL:                    /////
// - Start                            /////
// - Instance                         /////
// - Slot                             /////
// - Value                            ////
// - ValueList                        ///
// - static main                      //
////////////////////////////////////////

options {
 IGNORE_CASE = true;
}

PARSER_BEGIN(FIL)

import express.*;
import java.io.*;
import java.util.*;

/*****************************************************************************
** The class `FIL' implements a parser for the Frame Instance Language (FIL).
*****************************************************************************/

public class FIL {
 private static Hashtable/*FrameClass,Vector<Frame>*/ _frameClasses;
 private static CaseLibrary _caseLibrary;

 /*******************************************************************/
 // static FIL.getFrame : FrameClass x int -> Frame
 /*******************************************************************
 ** Looks up the <i>n</i>th instance of the given FrameClass. Note that
 ** the instance numbers here are as specified in the FIL file rather
 ** than the instance numbers frames are given internally.
 ** @param frameClass The frame class of the frame to look up.
 ** @param number The instance number (as specified in the FIL file) of
 **  the frame to look up.
 ** @returns The frame of that frame class which has that number.
 *******************************************************************/
```

```
public static Frame getFrame (FrameClass frameClass, int number) {
   Vector frames= (Vector)_frameClasses.get(frameClass);
   if (frames == null)
      _frameClasses.put(frameClass, frames= new Vector());
   if (number >= frames.size())
      frames.setSize(number+1);

   Frame frame= (Frame)frames.elementAt(number);
   if (frame == null) {
      try {
         frame= new Frame(frameClass, new Hashtable());
      } catch (InvalidSlot x) {
         System.err.println("This can never happen!");
      }
      frames.setElementAt(frame,number);
   }
   return frame;
}

/*******************************************************************/
// static FIL.main : String[] -> void
/*******************************************************************
** Runs the FIL parser on given input.
** @param argv The vector of command line arguments. Each argument is
** either an EXPRESS (<tt>.exp</tt>) file from which to construct the
** frame classes (do not include <tt>frame_classes.exp</tt> as that is
** read automatically) or an FIL (<tt>.fil</tt>) file for instances of
** those frame classes. At most one of the arguments can be a case
** library (<tt>.cas</tt>) file.
*******************************************************************/

public static void main (String args[]) throws ParseException {
   FIL parser= null;
   _caseLibrary= null; String filename= null;

   for (int a= 0; a < args.length; a++) {
      if (args[a].endsWith(".cas"))
         try {
            if (_caseLibrary != null) {
               System.err.println("There can only be one case library!");
               System.exit(1);
            }
            filename= args[a];
            _caseLibrary= new CaseLibrary(filename);
         } catch (FileNotFoundException f) {
            System.out.println("Case library \"" + filename + "\" not found - "
            + "will create it.\n");
            Frame.initialise();
            _caseLibrary= new CaseLibrary();
         }
   }//rof

   if (_caseLibrary == null) Frame.initialise();
```

```
    for (int a= 0; a < args.length; a++)
       if (args[a].endsWith(".exp"))
          Constraints.exjavaObjectsToFrames(new Express(args[a]), true);

    for (int a= 0; a < args.length; a++) {
       if (args[a].endsWith(".fil"))
          try {
             if (parser == null)
                parser= new FIL(new FileInputStream(args[a]));
             else
                parser.ReInit(new FileInputStream(args[a]));
             System.out.println("\nParsing " + args[a] + "...");
             _frameClasses= new Hashtable();
             parser.Start();
             System.out.println("Done.");
          } catch (FileNotFoundException e) {
             System.err.println(args[a] + " not found!");
             System.exit(1);
          }
    }

    if (parser == null) {
       System.out.println("No FIL file given - reading from standard input...");
       _frameClasses= new Hashtable();
       (new FIL(System.in)).Start();
    }

    Frame.generateFIL(true);

    System.out.println();

    if (_caseLibrary != null) {
       System.out.println("Saving case library \"" + filename + "\".");
       _caseLibrary.save(filename);
    }
  }
}

PARSER_END(FIL)


SKIP : {
   " " | "\t" | "\n" | "\r" |
   "(*" : COMMENT
}

<COMMENT> MORE : {
   <~[]>
}

<COMMENT> SKIP : {
   "*)" : DEFAULT
}
```

```
TOKEN : {
   <INSTANCE: "INSTANCE">
   |
   <END_INSTANCE: "END_INSTANCE">
   |
   <LOGICAL: "TRUE" | "FALSE" | "UNKNOWN">
   |
   <NULL: "NULL">
   |
   <IDENTIFIER: ["A"-"Z"] (["A"-"Z", "0"-"9", "_"])*>
   |
   <REAL: (["+", "-"])? <NUMBER> "." (<NUMBER>)?
          ("E" (["+", "-"])? <NUMBER>)?>
   |
   <NUMBER: (["0"-"9"])+>
   |
   <STRING: ("`" (~["`"])* "`") | ("\"" (~["\""])* "\"")>
}

////////////////////////////////
// Start::= (Instance)* <EOF> //
////////////////////////////////

void Start () : {
   Frame f;
}{
   (
      (f= Instance()) {
         if (_caseLibrary != null
         && f.getFrameClass().getName().equalsIgnoreCase("case_frame"))
            _caseLibrary.insert(new Case(f));
      }
   )*
   <EOF>
}

//////////////////////////////////////////////////////////
// Instance::= <INSTANCE> <IDENTIFIER> "#" <NUMBER> ";" //
//             (Slot)*                                  //
//             <END_INSTANCE> ";"                       //
//////////////////////////////////////////////////////////

Frame Instance () : {
   Token i,n;
   FrameClass c= null;
   Frame f= null;
}{
   <INSTANCE> (i= <IDENTIFIER>) "#" (n= <NUMBER>) ";" {
      try {
         c= FrameClass.lookup(i.image);
         f= getFrame(c,Integer.parseInt(n.image));
      } catch (FrameClassDoesNotExist x) {
         System.err.println("Frame class `" + i.image + "' does not exist!");
         System.exit(1);
      }
```

```
   } (Slot(f))* <END_INSTANCE> ";" {
      return f;
   }
}

////////////////////////////////////////
// Slot::= <IDENTIFIER> "=" Value ";" //
////////////////////////////////////////

void Slot (Frame f) : {
   Token s;
   Object v;
}{
   (s= <IDENTIFIER>) "=" (v= Value()) ";" {
      try {
         f.setSlotFiller(s.image,v);
      } catch (InvalidSlot x) {
         System.err.println("The slot `" + s.image + "' is invalid!");
         System.exit(1);
      }
   }
}

///////////////////////////////////////////
// Value::= ["+" | "-"] <NUMBER>         //
//        | <REAL>                       //
//        | <STRING>                     //
//        | <LOGICAL>                    //
//        | <NULL>                       //
//        | "?" <IDENTIFIER>             //
//        | <IDENTIFIER> "#" <NUMBER>    //
//        | <IDENTIFIER> "(" Value() ")" //
//        | <IDENTIFIER>                 //
//        | "{" (ValueList())? "}"       //
//        | "[" (ValueList())? "]"       //
///////////////////////////////////////////

Object Value () : {
   Token t,u;
   Object v;
   boolean n= false;
   UnorderedVector uo;
   OrderedVector o;
   DefinedType d= null;
}{
   ["+" | "-" {n= true;}] (t= <NUMBER>) {
      if (n) {
         return new Integer("-" + t.image);
         n= false;
      } else {
         return new Integer(t.image);
      }
   }
```

```
|
t= <REAL> {
   return new Double(t.image);
}
|
t= <STRING> {
   return t.image.substring(1, t.image.length()-1);
}
|
t= <LOGICAL> {
   if (t.image.equalsIgnoreCase("TRUE"))
      return new Logical(Logical.TRUE);
   else if (t.image.equalsIgnoreCase("FALSE"))
      return new Logical(Logical.FALSE);
   else
      return new Logical(Logical.UNKNOWN);
}
|
t= <NULL> {
   return new Null();
}
|
LOOKAHEAD(2) (t= <IDENTIFIER>) "#" (u= <NUMBER>) {
   try {
      return getFrame(FrameClass.lookup(t.image), Integer.parseInt(u.image));
   } catch (FrameClassDoesNotExist x) {
      System.err.println("Frame class `" + t.image + "' does not exist!");
      System.exit(1);
   }
}
|
LOOKAHEAD(2) (t= <IDENTIFIER>) "(" (v= Value()) ")" {
   try {
      d= DefinedType.lookup(t.image);
   } catch (DefinedTypeDoesNotExist x) {
      System.err.println("Defined type `" + t.image + "' does not exist!");
      System.exit(1);
   }
   return new DefinedTypeInstance(d,v);
}
|
(t= <IDENTIFIER>) {
   return new EnumSymbol(t.image);
}
|
"?" (t= <IDENTIFIER>) {
   return new Variable(t.image);
}
|
"{" {
   uo= new UnorderedVector();
} (ValueList(uo))? "}" {
   return uo;
}
```

```
    |
    "[" {
      o= new OrderedVector();
    } (ValueList(o))? "]" {
      return o;
    }
}

////////////////////////////////////
// ValueList::= Value ("," Value)* //
////////////////////////////////////

void ValueList (Vector v) : {
    Object e;
} {
    (e= Value()) {
      v.addElement(e);
    } ("," (e= Value()) {
      v.addElement(e);
    })*
}
```

This section has discussed the use of JavaCC as a language in which to specify a parser for the FIL language, and as a tool to generate the parser automatically as a Java program. The next section discusses the generation of frame browsers in HTML, which are just written out as FIL files augmented with HTML tags.

E.5 Generating HTML Frame Browsers

This section discusses the output from the constraint-understanding system, which is essentially the input EXPRESS model, augmented with HLCs. However, it does not simply write the model back out as EXPRESS, as the HLCs have no textual form at that level. Instead, it writes out the EXPRESS frames produced by parsing the model, along with the HLCs extracted from that model. This means that the output is essentially written out in FIL (c.f. Figures 6-2 to 6-5), but an idea to make it more easily browsable by humans was to write the output in HTML (Musciano and Kennedy [49]), augmenting the FIL code with hypertext links corresponding to frame references in slot fillers.

It should be noted that the only purpose of the external output of the constraint-understanding system at this stage of the research project is for humans to browse - to look at the HLCs the system has extracted from the input model. The constraint-understanding system is intended to be used as a building block for other systems in the future (in particular, a system for transferring the constraints on models for schema-to-schema mapping, as explained in Section 1.3.2) - its augmented models are ultimately intended for use by other systems rather than humans.

The system actually generates an FIL instance for every frame in its memory, which means that the frame class models and the cases can be browsed as well as the input EXPRESS model and the HLCs extracted from it. The internal frame instance numbers are used as the external frame instance numbers when the frames are written out as FIL. The FIL code can be augmented with HTML tags by setting a boolean flag. Figure E-1 illustrates by way of example how FIL files are augmented with HTML tags (which are highlighted in bold italics). This example comes from the FIL file of Figure 5-4.

As can be seen in Figure E-1, the system actually generates a separate FIL or HTML file for each frame class, with all the frames of that frame class contained in that file, as opposed to generating one big file for all the frames in the system.[11] So there are two files in Figure E-2: "`person.html`" for frames of frame class `person`, and "`department.html`" for frames of frame class `department`. The main reason for this is that each frame within an HTML file has an anchor, which hypertext links can refer to in a similar style to frame references in FIL (e.g. the hypertext link to `department#1` is written as "`department.html#1`").

Figure E-2 shows screen dumps of the two HTML files in Figure E-1, as rendered by the Netscape Navigator[12] 3.01 web browser.

11. This can, of course, violate FIL's scope rules for frame references and variables, so output FIL files would have to be concatenated to read them back in via the FIL parser.
12. "Netscape Navigator" is a trademark of Netscape Communications Corporation.

```
(********************)
(* File: person.html *)
(********************)

<HTML>
<H1>person</H1>
<HR>
<A NAME="1"></A>
<PRE>
INSTANCE <A HREF="entity_type.html#1">person</A>#1;
     <A HREF="explicit_attribute.html#1">name</A> = "Andrew Broad";
     <A HREF="explicit_attribute.html#2">age</A> = 23;
END_INSTANCE;
</PRE>
<HR>
<A NAME="2"></A>
<PRE>
INSTANCE <A HREF="entity_type.html#1">person</A>#2;
     <A HREF="explicit_attribute.html#1">name</A> = "Andrew Broad";
     <A HREF="explicit_attribute.html#3">department</A> (* no newline *)
     = <A HREF="department.html#1">department#1</A>;
END_INSTANCE;
</PRE>
</HTML>
```

```
(************************)
(* File: department.html *)
(************************)

<HTML>
<H1>department</H1>
<HR>
<A NAME="1"></A>
<PRE>
INSTANCE <A HREF="entity_type.html#2">department</A>#1;
     <A HREF="explicit_attribute.html#4">name</A> = "Computer Science";
END_INSTANCE;
</PRE>
</HTML>
```

Figure E-1. FIL augmented with HTML tags.

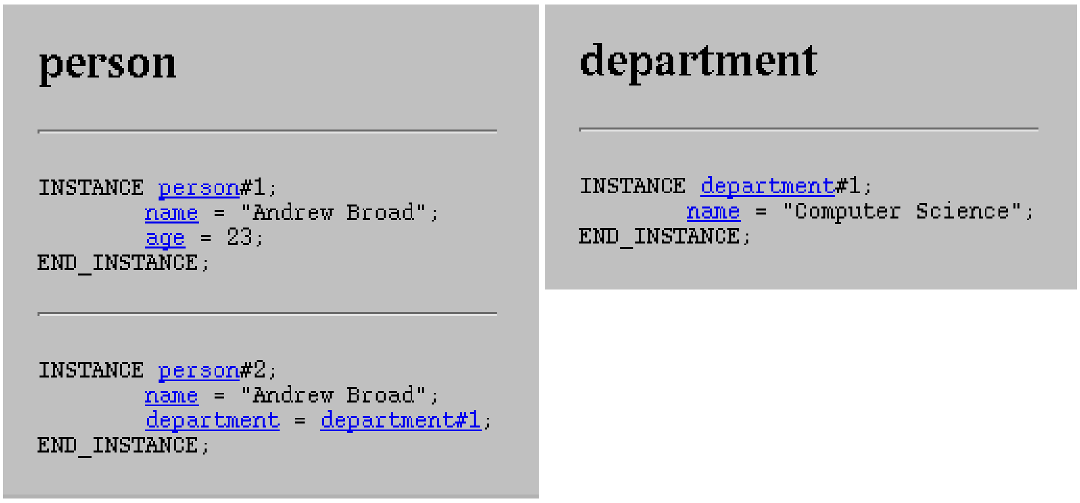

Figure E-2. Screenshots of the HTML files in Figure E-1, as rendered by Netscape Navigator 3.01.

Essentially, each FIL instance is augmented with HTML tags as follows:

- The frame has an anchor within that file for hypertext links to refer to (e.g. ***<A NAME="1">***).
- The name of the frame's frame class is tagged with a hypertext link to its frame class frame, allowing the user to browse the frame's meta-level frames. In Figure E-1, `entity_type#1` is the frame for the `person` frame class, and `entity_type#2` is the frame for the `department` frame class.
- Similarly, the name of each slot in the frame is tagged with a hypertext link to its slot frame. In Figure E-1, `explicit_attribute#1` is the frame for the `name` slot of `person`, `explicit_attribute#2` is the frame for the `age` slot of `person`, `explicit_attribute#3` is the frame for the `department` slot of `person`, and `explicit_attribute#4` is the frame for the `name` slot of `department`.
- Any frame references in the frame's slot fillers are tagged with hypertext links to the frames that they refer to (e.g. ***<A HREF="department.html#1">*** `department#1`***</A>***).

This appendix has described the implementation of the constraint-understanding system in Java, discussing the key decisions that were made at the implementation level (as opposed to the design level, which is covered elsewhere). Other languages and tools used to implement the constraint-understanding system were discussed, namely ExJava as the EXPRESS parser, JavaCC to generate the FIL parser, and HTML to augment the output FIL files with hypertext links to facilitate human browsing.