The CIDER Toolkit Reference Manual
Version 0.3.1

Anthony R. Jansen, Adrian Bickerstaffe
School of Computer Science and Software Engineering
Monash University
Victoria 3800
Australia

June 18, 2008

Abstract

Smart Diagram Environments (SDEs) are software applications that use structured diagrams to provide a natural visual interface that behaves as if the computer "understands" the diagram. Unfortunately, despite their potential usefulness, SDEs are not easy to build. This has motivated the creation of CIDER, a Java toolkit for building SDEs which greatly simplifies this task. CIDER is a generic component-based system which is designed to be easily embedded in Java applications. It provides automatic interpretation of diagrams as they are constructed and manipulated, structure preserving manipulation, and a powerful transformation system for specifying diagram manipulations and user interaction. CIDER's main innovation is its component-based approach to SDE development which provides substantially increased architectural flexibility to the application programmer. This manual provides instructions and explanations of CIDER's workings, and also includes a tutorial.

Contents

The CIDER toolkit was developed at the School of Computer Science and Software Engineering, Monash University, Australia, by Anthony Jansen, Kim Marriott, Bernd Meyer, Adrian Bickerstaffe and Aidan Lane.

The author wishes to thank the following people for their feedback on both the CIDER toolkit and this reference manual: Kim Marriott, Bernd Meyer, John Howse, Vivek Ramakrishna and Reginald Johnson.

Contact Information

Kim Marriott: *Kim.Marriott@infotech.monash.edu.au*
Bernd Meyer: *Bernd.Meyer@acm.org*

CIDER Website: `http://www.csse.monash.edu.au/~marriott/CIDER/`

1 Introduction

1.1 What is a Smart Diagram Environment?

Diagram Environments are software applications that allow users to create and manipulate structured diagrams via a natural visual interface. Smart Diagram Environments (SDEs) extend this functionality by behaving as if the computer "understands" the diagrams, by taking into account their structure and intended semantics [11].

Consider as an example, a software application designed to teach students about Finite State Automata (FSAs). It would be desirable for such an application to not only allow students to create and modify FSA diagrams, but also to perform other functions: visually demonstrate how to construct a FSA from a regular expression, determine whether a particular input string belongs to the language of the FSA, and construct a deterministic FSA from a non-deterministic FSA.

Unlike a standard graphics editor such as `xfig` the SDE would understand the structure of FSAs and automatically interpret the FSA diagram drawn by the user. In principle, the user constructs these diagrams through basic drawing operations from primitive graphical objects, such as lines and circles. Of course, in an ideal system additional support and high-level, composite graphical objects may be available where this simplifies the interaction. As an example consider the FSA shown in Figure 1. The SDE should identify the possible transitions and recognize structures such as the final state, represented by two concentric circles and the text 'S3'. Should the user move any one of the final state's components, the others should all move with it, preserving the final state's structure. Any transitions to or from the final state should also remained connected. The user should be warned if they have drawn an invalid FSA, for instance if there is an unlabeled transition or no start state.

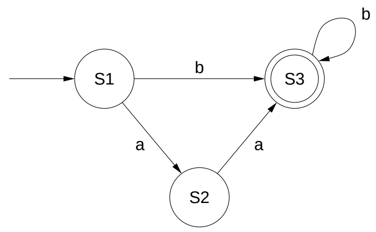


Figure 1: A simple finite state automaton.

While SDEs are potentially very useful, they are not easy to build. Luckily, there are a number of common functions that every SDE has to provide and implementing these subtasks can be supported by generic software tools. While several such tools have been created (such as DiaGen [13], GenEd [5], Recopla [12], Penguins [3] and GenGed [1]), they have limitations both in terms of their architecture and their capabilities. This has motivated the creation of Cider [6].

1.2 What is Cider?

Cider is a toolkit that simplifies the creation of SDEs. There are several functions that all SDEs should (ideally) provide:

- modelling of diagram components,
- automatic interpretation of diagrams as they are created and modified,
- structure preserving manipulation,
- a powerful transformation system for specifying diagram manipulations and user interactions, and
- automatic layout and diagram beautification.

Cider provides the first four capabilities as well as limited diagram layout, and its design allows the application to provide more comprehensive diagram layout and beautification. Diagrammatic languages are specified using constraint multiset grammars, while structure preserving manipulation is achieved via the use of a multi-way constraint solver.

Cider is written in Java, and has a component-based architecture. This means that it consists of generic components which the application wraps around, providing flexibility for the application programmer, and not imposing any restrictions on how the main application is programmed. The application programmer has complete control over the interface, and may choose not to use certain components, or possibly extend the toolkit's capabilities by providing additional components.

1.3 Constraint Multiset Grammars (CMGs)

Cider uses a variant of the Constraint Multiset Grammar (CMG) formalism for specifying diagrammatic languages [7]. Given a CMG specification, Cider can then perform bottom-up parsing of diagrams that are associated with that grammar. In this section we give a brief description of the Cider CMGs. Practical examples are given in the tutorial in Section 4.

CMGs are a kind of attributed multi-set grammar, and are specified in two parts: symbol definitions and production rules. Symbols have geometric and semantic attributes. Each symbol type can be *terminal* or *non-terminal*. Terminal symbols typically correspond to the primitive graphical objects in the diagram while non-terminal symbols are more complex objects built up from these.

Production rules are used to specify how non-terminal symbols are composed of other symbols. They are of the form

$$R_1, \ldots, R_m \ ::= \ S_1, \ldots, S_n \ \ \textit{where } C \ \ \textit{using } E$$

A rule is applied by testing if there is an assignment of symbols in the parse forest to $S_1, \ldots, S_n$, whose attributes satisfy the conditions C. If the evaluation is successful, the LHS symbols $R_1, \ldots, R_m$ are produced. The attribute values of these symbols are computed from the attribute values of the RHS symbols according to the expressions in E.

The production rule conditions, C, can include positive and negative context dependencies. This means that it is possible to have rules that are only applied if certain other symbols are also present, or alternatively only if certain other symbols are not present. Since negative context dependencies are present, Cider requires that the grammar specified be stratified (the Cider Compiler builds a symbol dependency graph to check this; see Appendix A for further details).

When a production rule is applied, the RHS symbols are said to be *reduced*. Symbols in the parse forest that have not been reduced are referred to as *root* symbols. The ability to produce multiple output symbols means that the Cider toolkit provides *unrestricted* (type 0) CMGs [9].

Cider also allows transformations to be specified. The basic idea behind a transformation is to change the parse forest of a diagram from one valid state to a different valid state. They are of the form

$$S_1, \ldots, S_n \ \ \textit{where } C \implies \textit{apply } (A_1, \ldots, A_m)$$

Transformations are initially evaluated by testing if, for some multiset of input symbols, $S_1, \ldots, S_n$, the attributes of the LHS symbols satisfy the condition C. In this respect they are similar to production rules, but unlike productions they are not applied automatically.

If the initial evaluation of a transformation is correct, then the LHS symbols are removed from the parse forest. Before each LHS symbol is removed it must be checked to see if it is used by another production further up in the parse forest, and if this is the case, then that production and its LHS symbols must be removed first. Also, if a LHS symbol is non-terminal, the entire sub-tree of this symbol is removed.

When the LHS symbols have been removed, the next step is to apply the actions $A_1, \ldots, A_m$. There are three types of action: add a new terminal symbol to the parse forest (and provide initial values for all of its attributes), modify the attribute of a symbol, and apply a particular production to a specified set of symbols.

1.4 Multi-way Constraint Solving

The CIDER toolkit incorporates a multi-way constraint solver. An important advantage in using a multi-way constraint solver is that it provides a structure preserving mechanism. Consider, for example, a condition on a production rule that requires two attribute values to be equal.

```
A.x == B.y
```

For any two candidate symbols `A` and `B`, the production can be successfully applied if the attribute values `A.x` and `B.y` are equal. However, if the above constraint is not enforced by a multi-way constraint solver and one of the attribute values is changed at a later stage, the production will become invalid (because the condition is no longer met). If, one the other hand, a multi-way constraint solver were used to enforce the condition, then if one of the attribute values was changed at a later stage, the multi-way constraint solver would ensure that constraint is not violated by also changing the other value. Thus, the production would remain valid and the structure of the diagram would be preserved.

The multi-way constraint solver used by CIDER is called QOCA [8]. QOCA operates over the domain of real numbers, and supports linear constraints (such as $x + 2y \leq 5$) but cannot handle constraints involving higher order polynomials (such as $x^2 - y^2 = 7$). However, for many diagrammatic languages, linear constraints are powerful enough to meet the application requirements. QOCA can also deal with both equalities and inequalities.

2 Using the CIDER Toolkit

2.1 System Requirements

The CIDER toolkit is written in Java, which is intended to be a platform independent language. As such, it can be run on many different architectures and with many different operating systems, including PC's running either Windows or Linux.

The CIDER toolkit requires at least Java version 1.4. Java version 1.4 can be downloaded from Sun's website, which can be found at the following URL:

`http://java.sun.com/j2se/`

Note that the SDK will need to be downloaded as the Runtime Environment, JRE, will not be sufficient for developing an application.marriott

This manual assumes that the developer of an application using the CIDER toolkit is a competent Java programmer, with a full understanding of the Java programming language (including aspects such as packages, inheritance, interfaces, APIs, static methods, event handling, etc.). Developers that do not have such a level of understanding are advised to seek reference material on Java, as features of the language which CIDER utilizes will not be explained in this manual.

2.2 Downloading

To use the CIDER toolkit, three files must be downloaded:

- the CIDER Compiler (JAR file)
- the CIDER Core Components (JAR file)
- the CIDER DTD file

These files can be downloaded from the CIDER website at the following URL:

`http://www.csse.monash.edu.au/~marriott/CIDER/`

The names of each of the files will include a version number. Ensure that the version number in the name of all three files is the same to ensure compatibility. Javadocs giving details of the Java classes contained in the Core Components JAR file are available at the CIDER website.

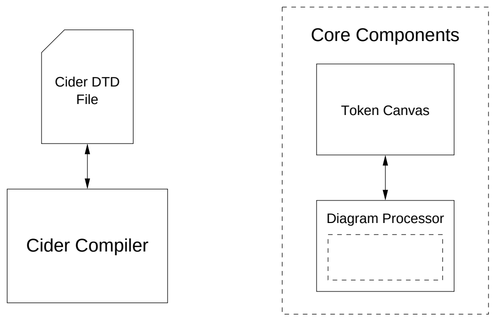


Figure 2: The CIDER toolkit when it is downloaded.

Figure 2 shows the components associated with these files. The Compiler and the DTD file will be used during the SDE's development to compile grammar specifications into Java code (allowing the resulting application to run faster), while the Core Components will be incorporated into the SDE application itself. The CIDER Compiler is a stand-alone Java application. The CIDERCore Components are a collection of Java packages with their own API that can be used in the creation of an application. These packages are just like the packages that come with Java. For example, to include a linked list in your Java program, you would use the `LinkedList` class defined in the package `java.util`. The CIDER Core Components are used in exactly the same way.

The CIDER toolkit is freely available under the terms and conditions of the GNU General Public License (see `http://www.gnu.org/copyleft/` for more information).

3 Creating a SDE Application using CIDER

3.1 Step 1: Create Grammar Specification Files

For the specific diagrammatic language that the application is being designed for, a grammar specification must be created. The CIDER toolkit uses a variant of the Constraint Multiset Grammars (CMG) formalism to specify diagrammatic languages. The specification consists of symbol definitions, production rules, and (optionally) transformation rules. The symbol definitions and production rules must all be placed in one file, and any transformation rules must be placed in a second file.

Sections 5 and 6 describe how grammar specification files are created, while Section 4 gives a tutorial with examples from a real diagrammatic language, providing details about the format used. Note that format discussed is the CIDER format (which is also the recommended format), but it is also possible for a grammar specification to be written in an XML format (see the DTD file that comes with CIDER for details on this format). Structurally, both formats are very similar, however the XML format is far more cumbersome.

3.2 Step 2: Compile Specification using the CIDER Compiler

This step involves compiling the grammar and transformations files with the CIDER Compiler in order to produce the language specific Java code that will be used by the CIDER Core Components, as shown in Figure 3.

To run the CIDER Compiler, the CIDER Compiler JAR file is needed. Also, the CIDER DTD file must be in the same directory as the specification files.

The CIDER Compiler accepts several command line arguments:

`-g`	*(required)*	Used to specify the name of the grammar file
`-t`	*(optional)*	Used to specify the name of the transformations file

(There are other less commonly used arguments that are available; refer to Appendix E for the full list of CIDER Compiler options and details of what they do.)

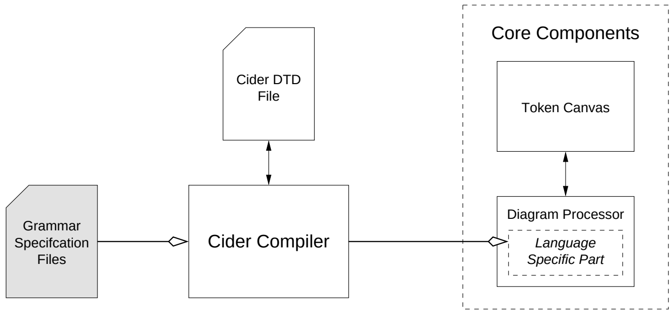


Figure 3: Compiling the Specification using the CIDER Compiler.

The CIDER Compiler is run using a command line similar to following (where the text contained inside square brackets is optional, and text in italics should be replaced with appropriate information):

`java -jar CiderCompiler-0.3.1.jar` `-g` *grammar_file*
`[-t` *transformations_file*`]`

The CIDER Compiler will first translate the grammar file and transformations file (if present) into an XML format. Thus, for each input file, a new file will produced whose name is the same as the original file, but with a `.xml` extension at the end. The XML files will then be processed to produce Java source code that is specific to the diagrammatic language that has been specified in the input files. This code belongs to the package `au.edu.monash.csse.tonyj.cider.interpreter` (and thus will be placed in the directory `au/edu/monash/csse/tonyj/cider/interpreter/` – see Java documentation on packages for an explanation of why this is required.)

One of the most important classes that will be created is the language specific *Interpreter* class, which controls most elements of diagram processing. The name of this class will be the grammar name followed by the string `Interpreter`. So for the finite state automata example, where the grammar name was `FSA`, the class will be `FSAInterpreter`. Also, for each symbol type that was defined, a Java class using the name of that symbol will also be created. So if, for example, the grammar includes a definition for a symbol type called `Circle`, a Java class called `Circle` will be generated.

Note that if there are any problems with the input files, the CIDER Compiler will halt with a message that details what the error is. If the error has occurred while the XML files are being processed, the error message may indicate what line of the XML file the error occurred on. When the CIDER Compiler produces the XML files, it includes comments that indicate on which lines of the original input file each part of the XML file corresponds to.

3.3 Step 3: Create Drawable Symbol Classes

For each type of symbol defined in the grammar specification, a Java class is created by the CIDER Compiler with the following properties:

- The class has the same name as the symbol that is specified.
- It belongs to the package `au.edu.monash.csse.tonyj.cider.interpreter`
- For each attribute, there is a `get_`*attribute_name*`()` method (for example, `get_radius()` for a `radius` attribute).
- For some attributes, there may also be a `set_`*attribute_name*`()` method (for example, `set_radius()` for a `radius` attribute). See Appendix B for details on which attributes will have this method.
- It has a constructor which accepts all of the attributes in the order they are specified (followed by the attributes of any inherited symbols if abstract symbols are involved).

However, most terminal symbols (and possibly some non-terminals) need a graphical representation, and the CIDER toolkit does not know how each such symbol should be drawn. Thus, a drawable version of these symbol classes must be created. Adding these drawable symbol classes extends the toolkit components as shown in Figure 4.

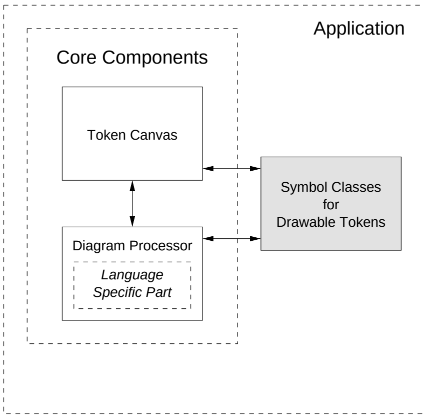


Figure 4: Adding the Drawable Symbol Classes.

Each drawable symbol class must do the following:

- belong to a named package (they cannot be part of the default package), where the fully qualified class name corresponds to that specified in the grammar file
- `extend` the corresponding symbol class produced by the CIDER Compiler
- include a constructor whose argument list is identical to that in the constructor of the symbol class produced by the CIDER Compiler, and whose values are passed immediately to the constructor of that terminal symbol class via the call `super()` (the constructor should do nothing else)
- `implement` the `DrawableToken` interface which is part of the package `au.edu.monash.csse.tonyj .cider.canvas`; this involves implementing a `draw()` method and a `contains()` method (see the Javadocs on the CIDER Core Components for more information on the `DrawableToken` interface).

3.4 Step 4: Create Java Methods required by the Grammar

If the grammar or transformations specified require the use of Java methods that are not part of the core Java API, these methods must be created inside the appropriate Java classes. The methods must be `static` Java methods that do not throw exceptions, and the classes that contain them must belong to a named package (they cannot be part of the default package). (For more information on `static` methods or Java packages, consult the Java language specification.) Adding these Java Methods extends the toolkit components as shown in Figure 5.

3.5 Step 5: Compile code with the Java Compiler

The source code produced by the CIDER Compiler in Step 2, as well as the source code produced in Steps 3 and 4 can now be compiled using the Java compiler (the `javac` command).

Make sure that all source code is in an appropriate directory relative to the package that it belongs to (see the Java documentation for information on compiling classes that belong to a package). Also ensure that the classpath contains the CIDER Core Components JAR file. Don't forget that the classpath will probably need to include the current directory, which is denoted by a single dot (`.`). (See Java documentation for more information on classpaths.)

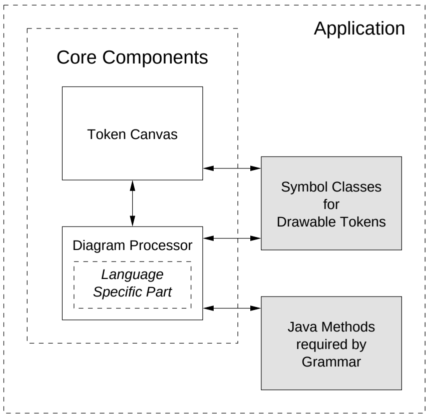


Figure 5: Adding the required Java Methods.

The source code can then be compiled using a command line similar to the following (where the text contained inside square brackets is optional, and text in italics should be replaced with appropriate information):

`javac [-classpath` *classpath*`]` *source_files*

where *source_files* should include the source code produced by the CIDER Compiler (which should be specified as `au/edu/monash/csse/tonyj/cider/interpreter/*.java`), as well as classes that implement user defined functions defined in the grammar and/or transformations specifications (if any), and drawable versions of the terminal symbol classes.

By doing this step before the rest of the application is written, it should be possible to isolate and correct compile time errors more easily.

3.6 Step 6: Create the rest of the Application

Now, the rest of the application can be written around the CIDER Components as shown in Figure 6.

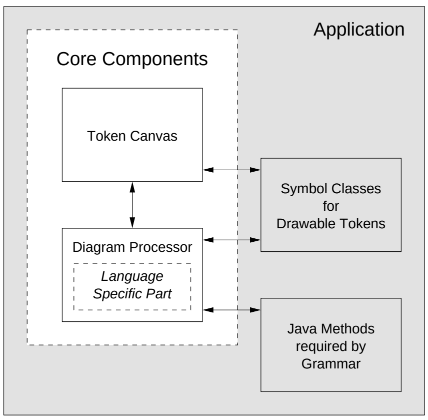


Figure 6: Creating the Rest of the Application.

Typically, the main interaction between the CIDER Components and the application will be via an `InterpretedTokenCanvas` (which can be found in the package `au.edu.monash.csse.tonyj.cider.canvas`). This is a special type of `TokenCanvas` that works specifically with a CIDER `Interpreter` (which is the main class that does the diagram processing). In the constructor for the `InterpretedTokenCanvas`, one needs to provide the canvas dimensions (in pixels) as well as an `Interpreter` for the specific diagrammatic language being used. The following is an example for the FSA specification.

```
InterpretedTokenCanvas canvas;
canvas = new InterpretedTokenCanvas(600, 400, new FSAInterpreter());
```

From the canvas, one can access the interpreter using the `getInterpreter()` method. **Note:** The current Cider implementation can run at most one `InterpretedTokenCanvas` (and therefore, one interpreter) at a time. Attempts to load more than one interpreter will result in a clash in the namespace of the Cider run-time environment. This will present as an uncaught run-time exception.

The `InterpretedTokenCanvas` is Java Swing component (extending `JPanel`), and thus should be treated accordingly. For example, if the attributes of symbols in the parse forest have changed resulting in the diagram appearing different, then a call to the `repaint()` method should be made so that the change is updated in the display. Mouse events that occur in the canvas can be listened for using the normal event classes for doing this, and the canvas can be passed to a layout manager in the normal way.

New terminal symbols can be added to the canvas by creating drawable symbol objects (using the classes created in Step 3), and passing them to the canvas using the `add()` method. They can be removed using the `remove()` method. Note that non-terminals are only created internally by CIDER, and so cannot be explicitly added by the application. The attributes of symbols can be modified using the `set_`*attribute_name*`()` methods (although these methods may not be publicly available for many non-terminal symbol attributes, as their values may be enforced by grammar rules and thus cannot be freely modified – see Appendix B for more detail).

Since the diagram processing is handled by the CIDER components, the application programmer does not have too much to worry about in this regard. However, if other parts of the application need to be aware of changes that are occurring to the diagram structure, the `GrammarSymbolListener` interface can be implemented. By adding a listener to the interpreter, the listener class can be informed of events such as new symbols being added to the parse forest, symbols being removed from the parse forest, and symbol attributes being modified. This information is passed inside the `GrammarSymbolEvent` class, and the system operates using standard Java event handling methods.

By default, production rules are automatically processed by CIDER during diagram manipulation. However this option can be turned off if desired. It can sometimes be useful to do this temporarily, such as when changing multiple attribute values at once. For example, consider changing two attributes, the `mid_x` and `mid_y` values of a terminal symbol referred to by the variable `sym` as shown in the following Java code.

```
sym.set_mid_x(newX);
sym.set_mid_y(newY);
```

What may happen here is that the first change (to `mid_x`) invalidates part of the parse forest forcing the Interpreter to remove it, whereas after the second change (to `mid_y`), that part of the parse forest needs to be recreated again. This means unnecessary extra work for the Interpreter. If the `InterpretedTokenCanvas` has the variable name `canvas`, the following additions can be made to the Java code.

```
canvas.getInterpreter().setAutoForestChecking(false);
sym.set_mid_x(newX);
canvas.getInterpreter().setAutoForestChecking(true);
sym.set_mid_y(newY);
```

This will ensure that parse forest is only rechecked once the second change (to `mid_y`) is made, possibly preventing the Interpreter from having to do redundant work. (Note that automatic forest checking needed to be reactivated *before* the final attribute modification.)

While productions are usually processed automatically (unless this option is turned off), transformations must be explicitly called in order for them to be applied. This is generally done with one of the two methods `applyTransformation()` or `applyTransformationInParallel()`. Both of these methods take a single argument, an integer that identifies the transformation to apply, and both return a boolean value that indicates whether the transformation was successfully applied. If the transformation could not be applied, the diagram will remain completely unchanged.

This integer that specifies which transformation is to be applied is defined in the language specific Interpreter class (such as `FSAInterpeter`) by a static constant. The name of the constant is the string TRANSFORMATION_ followed by the specified name of the transformation in the input file. This convention holds regardless of whether the transformation is a basic transformation, a disjunction, or a compound transformation. So if, for example, the FSA grammar includes basic transformation called CreateFinalState, its corresponding integer value in the application code is accessed through the constant FSAInterpreter.TRANSFORMATION_CreateFinalState.

See the Javadocs on the CIDER Core Components for more information on how to use the CIDER components in an application, and/or look at the example applications on the CIDER website.

Remember that when the complete application is compiled and run, its classpath must have access to the CIDER Core Components JAR file, and any classes produced in Steps 2, 3 and 4.

4 Grammar Specification Tutorial

The purpose of this section is to introduce how constraint multiset grammars are specified in the CIDER toolkit. In this example, we consider the task of using CIDER to build a SDE for Finite State Automata (FSAs). Note that grammar is specified in an ordinary text file which can have any name (so we can but the FSA grammar specification in a file called fsa.grammar).

4.1 Symbol Definitions

Let's begin by focusing on a particular area: the creation of FSA states. FSA states are composed of circles and text strings. Normal states consist of a text string in the centre of a single circle, while final states consist of a text string in the centre of two concentric circles. It is clear that we need two types of terminal symbols; a Circle symbol class and a Text symbol class.

```
Circle (terminal);
Text (terminal);
```

The string (terminal) indicates that the two symbol classes represent terminal symbols. Each symbol will also require attributes, so the above specification needs further additions.

```
Circle (terminal)
   radius:double, mid_x:double, mid_y:double;
Text (terminal)
   label:java.lang.String, mid_x:double, mid_y:double;
```

Each symbol has three attributes given in a comma separated list. The attributes are specified using the format *name:datatype*, where *datatype* is any primitive Java data type or a fully qualified classname.

Two non-terminal symbol classes will also need to be created to represent the FSA states. These will also require several attributes.

```
NormalState (non-terminal)
   radius:double, mid_x:double, mid_y:double, label:java.lang.String;
FinalState (non-terminal)
   radius:double, mid_x:double, mid_y:double, label:java.lang.String;
```

At this point there seems to be some unnecessary duplication. Both NormalState and FinalState use the exact same attributes, and in some circumstances we will want to treat symbols of both classes simply as generic states. In this instance then, it makes sense to use an abstract symbol class.

```
State (abstract)
   radius:double, mid_x:double, mid_y:double, label:java.lang.String;
```

Now we can modify our two non-terminal symbols to inherit from the State symbol class, which will make the specification both simpler, and more flexible.

```
NormalState (non-terminal) : State;
FinalState (non-terminal) : State;
```

Note that the inheritance syntax is similar to that used in C++ (making use of the `:` character).

One further thing needs to be specified in the symbol class definitions. CIDER does not know how to draw the terminal symbols, so the application developer must provide classes that do the drawing. So for each terminal symbol, a fully qualified class name must be specified. This class must extend the class produced by the CIDER Compiler. The name of the class must appear within square brackets (`[` and `]`) and must be placed immediately after the `(terminal)` string. For our example, the `Circle` and `Text` definitions need to be added to.

```
Circle (terminal) [fsa_editor.DrawableCircle]
   radius:double, mid_x:double, mid_y:double;
Text (terminal) [fsa_editor.DrawableText]
   label:java.lang.String, mid_x:double, mid_y:double;
```

Note that in this case, both of the classes specified belong to the package `fsa_editor`.

4.2 Production Rules

With our symbol classes defined, we can now specify the production rules for creating FSA states. All production rules in CIDER are specified using the `production` keyword in the following manner.

```
:production {
   ...
}
```

As there is no text in front of the `:` character, this format specifies an anonymous production. Productions can optionally be named by providing a unique name in front of the `:` character.

We will first create a production for a normal state. A normal state production will need a `Circle` and a `Text` symbol as input, and will produce a `NormalState` symbol as output. Adding to our production definition, we get:

```
:production {
   n:NormalState ::= c:Circle, t:Text
   ...
}
```

The output symbol is on the left of the `::=` string, with the input symbols being on the right (in a comma separated list). Each symbol is given a unique name so that it can be referred to in the production. The resulting format for each symbol specified is *name:symbol_class*.

Next we need to consider the conditions that must be met for a normal state to be created. A circle and text string should only form a normal state if they share the same mid point. So we must add a constraint set (using the `where` keyword), and add the required constraints.

```
:production {
   n:NormalState ::= c:Circle, t:Text
   where {
      c.mid_x == t.mid_x;
      c.mid_y == t.mid_y;
   }
   ...
}
```

Finally, the output `NormalState` symbol has several attributes (all inherited from the abstract `State` symbol class) that need to be set. Setting these attributes involves not only initializing them, but specifying one way constraints (using the `set` keyword). This ensures that if any attributes of the input symbols are modified later, any output symbol attributes that depend on them are also updated (for example, if the text label is changed, the label of the normal state should be modified accordingly).

```
:production {
   n:NormalState ::= c:Circle, t:Text
   where {
      c.mid_x == t.mid_x;
      c.mid_y == t.mid_y;
   }
   set {
      n.label := t.label;
      n.radius := c.radius;
      n.mid_x := c.mid_x;
      n.mid_y := c.mid_y;
   }
}
```

Note that *all* output symbol attributes must be accounted for.

The production rule for a final state will be very similar to the above production rule for a normal state. The main difference is that the final state is composed of *two* circles and a text string, resulting in additional constraints being required.

```
:production {
   f:FinalState ::= c1:Circle, c2:Circle, t:Text
   where {
      c1.mid_x == t.mid_x;
      c1.mid_y == t.mid_y;
      c2.mid_x == t.mid_x;
      c2.mid_y == t.mid_y;
      c1.radius > c2.radius;
   }
   set {
      f.label := t.label;
      f.radius := c1.radius;
      f.mid_x := c1.mid_x;
      f.mid_y := c1.mid_y;
   }
}
```

4.3 Positive and Negative Context Tests

Despite the fact that our two production rules will correctly create normal and final FSA states, there is still a problem. Consider the case where there are two `Circle` symbols and a `Text` symbol, all sharing a common mid point. Clearly these symbols are meant to represent a final state. However, if the normal state production rule is used first, the symbols will be interpreted as representing a normal state that shares the same mid point as an additional circle. Clearly this is not what is intended.

This problem can be overcome by adding a further constraint to the normal state production rule, one that involves a *negative context* test. Note that the negative context constraint has its own constraint set.

```
:production {
   n:NormalState ::= c:Circle, t:Text
   where {
      c.mid_x == t.mid_x;
      c.mid_y == t.mid_y;
      not_exists c2:Circle
      where {
         c2.mid_x == t.mid_x;
         c2.mid_y == t.mid_y;
      }
   }
   ...
}
```

A negative context constraint is a condition that requires that a certain type of symbol *must not* be present in the parse forest. The addition of this negative context constraint to the production rule for a `NormalState` symbol ensures that the production will never be applied in situations where a final state production is more appropriate.

Note that it is also possible to create productions that include a *positive context* test. These work in the same way as a negative context test, except that they require a certain type of symbol to be present in the parse forest, rather than absent. For example, consider the production for a transition, which consists of an arc between two states. The `Arc` symbol contains the attributes `start_x` and `start_y` to indicate its starting point, and `end_x` and `end_y` to indicate its end point (where all these attributes have the datatype `double`). We need not be concerned here with the attributes of the `Transition` symbol type.

```
:production {
   t:Transition ::= a:Arc
   where {
      exists s1:State
      where {
         fsa_editor.Functions.OnCircle(a.start_x, a.start_y,
                                  s1.mid_x, s1.mid_y, s1.radius) == true;
      }
      exists s2:State
      where {
         fsa_editor.Functions.OnCircle(a.end_x, a.end_y,
                                  s1.mid_x, s1.mid_y, s1.radius) == true;
      }
   }
   ...
}
```

The two `State` symbols, `s1` and `s2`, must be defined as positive context symbols rather than input symbols. This is because input symbols are reduced when a production is applied, and once a symbol is reduced it cannot be used as an input symbol in another production. This would not be appropriate for the states in a transition production, as each state in a FSA is allowed to have any number of transitions connected to it.

4.4 Using Static Java Functions

The above production for transitions also includes two calls to a static Java function. The function is called `OnCircle`, and determines if a given point is located on the circumference of a specified circle. The function must be static, public, and not throw any exceptions. It must also be defined in a public class that belongs to a package (in this case, the package name is `fsa_editor`). Note the use of the `==` in the production rule to compare the output of the function with the constant `true`.

The following would be a valid implementation of the `Functions` class that contains the `OnCircle` method.

```
package fsa_editor;

public class Functions {

   public static boolean OnCircle(double x1, double y1,
                                  double x2, double y2, double radius) {
      double distance;
      distance = Math.sqrt(((x2 - x1) * (x2 - x1)) + ((y2 - y1) * (y2 - y1)));
      return (distance == radius);
   }
}
```

Note: Special care must be taken when writing static Java functions for use with CIDERṠhould a run-time exception occur in a called function, Cider will not report an error and will continue running. This can misleadingly appear as if the Java function was not called. It is wise to include exception handling in the static Java functions, even if exceptions are simply caught and printed.

4.5 Using Multi-Way Constraints for Structure Preservation

Let us now return to the previously defined production for final states. Unfortunately, the production constraints as they are currently specified are merely tests which are not enforced. This means that if a final state is created, and then the text is moved to a new location, the production is simply invalidated and the final state will no longer exist. However, it is preferable in such a situation for the constraints to be enforced, so that when the text is moved, the associated circles move with it, thus preserving the structure of the final state.

In order to obtain such structure preserving behaviour, a multi-way constraint solver must be used. CIDER incorporates a multi-way constraint solver called QOCA [8]. QOCA is a linear constraint solver that operates over the domain of real numbers, and can deal with both linear equalities and inequalities (see Section 1.4).

QOCA involves the use of constraint variables, solver constraints, and constraint expressions in the specification. The constraint variables are specified as the datatypes for the relevant symbol attributes, the solver constraints are used in the constraint sets, while the constraint expressions appear when setting attribute values. We now consider how they are applied to the final state production.

Firstly, all symbol attributes that are to be used in a multi-way constraint must have their datatype changed to a constraint variable. This is done using the `@cv` command, with the stay and edit weights of the variable following this string as two (comma separated) positive floating point values inside a pair of square brackets.

Users cannot modify constraint variables directly. Instead, they can only suggest a new value for the variable. The multi-way constraint solver will then check if this new value can be allocated or not, and this may also involve changing the values of other constraint variables depending on the constraints in the solver. The value of the stay and edit weights of each constraint variable will determine which variables will have their value changed when a new value is suggested (see Section 5.3.3 for further details).

Since the multi-way solver only operates on the domain of real numbers, attributes that represent, for example, strings, cannot be constraint variables. Updating our example, we get the following.

```
Circle (terminal) [fsa_editor.DrawableCircle]
   radius:@cv[100, 1000], mid_x:@cv[10, 1000], mid_y:@cv[10, 1000];
Text (terminal) [fsa_editor.DrawableText]
   label:java.lang.String, mid_x:@cv[10, 1000], mid_y:@cv[10, 1000];
State (abstract)
   radius:@cv[100, 1000], mid_x:@cv[10, 1000], mid_y:@cv[10, 1000],
   label:java.lang.String;
NormalState (non-terminal) : State;
FinalState (non-terminal) : State;
```

Next, we replace all the appropriate conditions in the constraint set of the final state production rule with solver constraints. This is done using the `@sc` command, which is followed by a pair of parentheses that takes two (comma separated) arguments. The first argument is a tolerance amount, and is a positive floating point value. The tolerance amount allows us to specify that, rather than requiring exact positioning, if a text string is placed within 3 pixels of the centre of a circle, that is good enough to successfully meet our condition. This is useful in GUI applications where it would be difficult for users to place objects with absolute precision. The second argument is the constraint string. Each constraint variable referred to in the string appears inside a pair of square brackets. (See Section 5.4.4 for details of the format used.)

```
:production {
   f:FinalState ::= c1:Circle, c2:Circle, t:Text
   where {
      @sc(3.0, "-[c1.mid_x] + [c2.mid_x] == 0.0");
      @sc(3.0, "-[c1.mid_y] + [c2.mid_y] == 0.0");
      @sc(3.0, "-[c1.mid_x] + [t.mid_x] == 0.0");
      @sc(3.0, "-[c1.mid_y] + [t.mid_y] == 0.0");
      @sc(3.0, "[c1.radius] + -[c2.radius] <= -1.0");
   }
   set {
      ...
   }
}
```

Finally, the attribute settings of the final state symbol need to be changed. This is done by setting the attributes as being equal to a constraint expression, by using the `@ce` command. This command is followed by a pair of parentheses containing a constraint expression string. Each attribute setting that involves a constraint expression is then passed to the multi-way solver where the constraint *attribute* == *expression* is enforced. Note that the setting of the `label` attribute is unchanged, as it is not a constraint variable.

```
:production {
   f:FinalState ::= c1:Circle, c2:Circle, t:Text
   where {
      ...
   }
   set {
      f.label := t.label;
      f.radius := @ce("[c1.radius]");
      f.mid_x := @ce("[c1.mid_x]");
      f.mid_y := @ce("[c1.mid_y]");
   }
}
```

4.6 Transformation Rules

Now we turn to an example of specifying a transformation. Transformations must be specified in a different file to the rest of the grammar. As with the grammar file, the transformations file is also a simple text file which can have any name (so we can use a file called `fsa.transformations`).

A transformation specifies a manipulation that should be made to the diagram if certain conditions are met. Transformations are very similar to production rules, with the main difference being that transformations happen at a single moment in time and are then forgotten, whereas the conditions involved with a production rule must be continually monitored and enforced.

For our example transformation, we will consider the processing of an input string in a FSA diagram. Specifically, whether the input string can be processed by any of the transitions that start from the state where it currently is. If it can be processed, the input string should be removed from its current state, and a new input string should be added to the destination state of the transition. The new input string will of course be shorter than the old one (having the transition symbol removed from the front). This is illustrated in Figure 7.

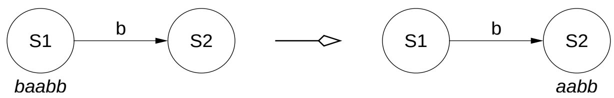


Figure 7: Processing an Input String in a FSA.

This tutorial has only given a few examples of symbol definitions and production rules used for FSA diagrams, without providing a complete grammar. To make our transformation rule example clearer, we will need to know about two more symbol definitions. Note that for both symbol types, some of the attributes will be instances of the `State` symbol type.

```
InputString (non-terminal)
   label:java.lang.String, state:State;
Transition (non-terminal)
   from:State, to:State, transition:java.lang.String;
```

We also need to know about a production rule called `pInputString`, which is used to create `InputString` symbols. This occurs when a `Text` symbol is placed about 12 pixels beneath a `State` symbol.

```
pInputString:production {
   i:InputString ::= t:Text
   where {
      exists s:State
      where {
         @sc(3.0, "[t.mid_x] + -[s.mid_x] == 0.0");
         @sc(3.0, "[t.mid_y] + -[s.mid_y] + -[s.radius] == 12.0");
      }
   }
   set {
      i.label := t.label;
      i.state := s;
   }
}
```

Now we can look at creating our transformation rule. All transformations must have a unique name, and are specified using the `transformation` keyword (in a manner similar to the use of the `production` keyword). We will call our transformation rule `TwoStateProcessing`.

```
TwoStateProcessing:transformation {
   ...
}
```

While a production rule would next specify input and output symbols, in a transformation rule we only specify input symbols in a comma separated list. If the conditions of the transformation are met, all input symbols will be removed from the diagram. For our example, we have only a single input symbol, an `InputString`.

```
TwoStateProcessing:transformation {
   i:InputString
   ...
}
```

We now defined the conditions that must be met in much the same manner as is done for a production rule. The main difference is that, because transformations only happen at a single point in time (unlike production rules whose conditions may be enforced afterwards), there is no need to add any constraints to a constraint solver. Thus, solver constraints are not used when specifying transformations.

The first condition that must be met, is that there must be a `Transition` that matches the start of the `InputString`.

```
TwoStateProcessing:transformation {
   i:InputString
   where {
      exists t:Transition
      where {
         fsa_editor.Functions.startsWith(i.label, t.transition) == true;
         ...
      }
   }
   ...
}
```

We check this by passing the `InputString` label and the `Transition` string to a function we have created (or will create), which will return either true or false. Note that the function is called `startsWith`, is inside a class called `Functions` which belongs to the package `fsa_editor`.

Next we have to ensure that the transition is relevant - that is, that the transition starts at the same state that the input string is associated with.

```
TwoStateProcessing:transformation {
   i:InputString
   where {
      exists t:Transition
      where {
         fsa_editor.Functions.startsWith(i.label, t.transition) == true;
         exists s1:State
         where {
            t.from == s1;
            i.state == s1;
         }
         ...
      }
   }
   ...
}
```

Finally, we will also need to identify the state that the transition goes to.

```
TwoStateProcessing:transformation {
   i:InputString
   where {
      exists t:Transition
      where {
         fsa_editor.Functions.startsWith(i.label, t.transition) == true;
         exists s1:State
         where {
            t.from == s1;
            i.state == s1;
         }
         exists s2:State
         where {
            t.to == s2;
         }
      }
   }
   ...
}
```

If the input conditions are met, we then need to carry out certain actions in order to modify the diagram (aside from removing the input symbols, which does not need to be explicitly specified). These actions are specified inside using the `apply` keyword, which appears at the end of the transformation rule.

```
TwoStateProcessing:transformation {
   i:InputString
   where {
      ...
   }
   apply {
      ...
   }
}
```

Although we wish to create a new `InputString` symbol, we cannot create it directly because it is a non-terminal symbol. Instead we must first create the terminal symbols that it requires, and then call the appropriate production rules to create the terminal symbol for us. Even then it will only apply the productions if the resulting diagram will remain syntactically valid. Thus it is not just the previously specified conditions that must be met for the transformation to be carried out; it must also be possible to apply all of the actions that are specified.

Since `InputString` symbols are created from `Text` symbols, our first action is to create such a symbol using the `(nts)` command (where `nts` stands for new terminal symbol).

```
TwoStateProcessing:transformation {
   ...
   apply {
      (nts)   text:Text(fsa_editor.Functions.tail(i.label, t.transition), s2.mid_x,
                       fsa_editor.Functions.yOffset(s2.mid_y, s2.radius));
      ...
   }
}
```

After the `(nts)` string, the new symbol has been specified with a unique name, and then all of its attributes have been initialized in a comma separated list that is placed between parentheses. The order of the arguments in the list corresponds to the order in which the attributes were defined in the symbol definition. Note that here also, two of the attributes were set using calls to functions.

Our final action is apply the production `pInputString` using the `(ap)` command, in order to create an `InputString` symbol. The `(ap)` command is followed by the name of production rule to be applied, and then a comma separated list of symbol mappings that shows which symbols in the transformation rule should be mapped to which input or existential symbols in the `pInputString` production rule.

```
TwoStateProcessing:transformation {
   ...
   apply {
      (nts)   text:Text(fsa_editor.Functions.tail(i.label, t.transition), s2.mid_x,
                        fsa_editor.Functions.yOffset(s2.mid_y, s2.radius));
      (ap)    pInputString using(text -> t, s2 -> s);
   }
}
```

In this case the newly created text object is passed to production rule's input symbol, while the destination state of the transition is passed to the production rule's existential symbol. Mapping of symbols allows you to remove any possible conflicts where there may be multiple candidate symbols that could be used by a given production.

5 Format for the Grammar File

For any diagrammatic language for which an editor is to be created, a grammar must be provided. The CIDER toolkit uses a variant of the Constraint Multiset Grammar (CMG) formalism, which is a type of attributed multiset grammar. It receives the grammar specification in a text file. The entire grammar specification for the given diagrammatic language must be contained within a single file. Any transformations required must go in a separate file (see Section 6). This section describes the format for the grammar file, using a grammar specification for Finite State Automata (FSAs) as an example.

5.1 Some Conventions

Comments can be placed in the grammar file using the `#` character. This character can appear anywhere in the file, and any text that follows it on the same line is ignored. Note that any text that precedes the `#` character on a given line is still processed. Also, the `#` character does not represent a comment if it appears inside a quoted string.

Quoted Strings are strings that appear inside quotation marks. The quotation marks used to delimit a given string may be either singles quote marks (`'`) or double quote marks (`"`). Thus, single quote marks may safely appear in a string that is delimited by double quote marks, and vice versa. If a string must contain a quote mark of the same type as its delimiters, it must be preceded by a backslash character (i.e. use `\"` to represent a double quote mark, or `\'` to represent a single quote mark). Note that quoted strings may not cross multiple lines.

Proper Names are strings where the first character is a letter and the remaining characters are either letters, digits, or the underscore (`_`) character. Note that all naming is case sensitive.

5.2 Grammar File Contents

The contents of the grammar file can be broken up into three sections:

1. the grammar name
2. the symbol definitions
3. the production rules

These three sections must appear in the grammar file in the order listed here.

The simplest of these sections is the specification of the grammar name, which is specified by the `grammar` keyword. An example of a grammar name declaration is:

```
grammar = "FSA";
```

Note that the grammar name must be a proper name, and appear as a quoted string. Also note that this declaration is concluded with a `;` character.

5.3 Symbol Definitions

The symbol definitions are specified using the `symbol_definitions` command (where the contents of this command must appear between the `{` and `}` characters). Inside this command any number of symbol definitions can be specified.

To understand how symbol definitions are specified, consider the following example.

```
Text (terminal) [fsa_editor.DrawableText]
   label:java.lang.String, mid_x:double, mid_y:double;
```

The definition begins with the name of the symbol, which must be a proper name. In this example, the name of the symbol is `Text`. For each symbol that is defined, the CIDER toolkit will produce a Java class using the specified name. The class produced will belong to the package `au.edu.monash.csse.tonyj.cider.interpreter` (as will all classes produced by the CIDER Compiler). Due to this, there are restrictions of the names that can be given in a symbol definition, both in the form of illegal names (strings that cannot be used), and illegal substrings within names (for example, if `Undo` is an illegal substring, then `Undo`, `MyUndo` and `Undo7` are all illegal names). The list of restrictions is:

Illegal Symbol Names:	`Error`, `Log`, `ParseForest`, `RegExpTerm`
Illegal Symbol Substrings:	`Cider`, `Constraint`, `Interpreter`, `Production`, `Symbol`, `RegularExpression`, `Transformation`, `Undo`

There are also two reserved symbol names that cannot be defined by the user (although they can be used in the grammar – see Section 7 on force directed graph layout). They are `GraphNode` and `GraphEdge`.

Following the name, the type of symbol appears inside a pair of parentheses. There are four possible symbol types: `terminal` for a terminal symbol type in the grammar (as in the above example); `non-terminal` for non-terminal symbol types in the grammar; `start`, which indicates it is the start symbol type in the grammar (this is just a special type of non-terminal symbol - there should only be one such symbol type per grammar); and `abstract`, which will be discussed shortly.

Following the symbol type in the above declaration is the *Drawable Class* attribute, which is the name of a Java class appearing inside square brackets. This component of a symbol definition typically applies to symbols whose type is `terminal`, and is concerned with how these symbols are drawn. In the application that is being created, the diagram will be manipulated using a Java object called `TokenCanvas` (more specifically an `InterpretedTokenCanvas` which inherits from `TokenCanvas`). The tokens in `TokenCanvas` are objects that implement the `DrawableToken` interface. Objects which implement this interface contain a method telling the `TokenCanvas` how to draw them. However, the Java classes that are produced by the CIDER toolkit to represent the symbols do not implement the `DrawableToken` interface, as it is not possible for the CIDER toolkit to known how all the various graphical symbols should be drawn. Instead, this task must be carried out by the person who is creating the application. A new class, identified by the Drawable Class attribute, must therefore be created. This class should

- `extend` the class produced by the CIDER toolkit for that symbol,
- `implement` the `DrawableToken` interface so that the `TokenCanvas` will be able to draw it,
- include a constructor whose argument list is identical to that in the constructor of the symbol class produced by the CIDER Compiler, and whose values are passed immediately to the constructor of that terminal symbol class via the call `super()` (the constructor should do nothing else), and

- belong to a Java package.

Note that the Drawable Class attribute must include the fully qualified name of this new class.

In the current example, the class that was (or will be) created is called `DrawableText`, and is included in a package called `fsa_editor`. The Java code for the class declaration will look like the following.

```
package fsa_editor;

import au.edu.monash.csse.tonyj.cider.interpreter.Text;
import au.edu.monash.csse.tonyj.cider.canvas.DrawableToken;

public class DrawableText extends Text implements DrawableToken {

   // Constructor
   public DrawableText(java.lang.String label, double mid_x, double mid_y) {
      super(label, mid_x, mid_y);
   }

   public void draw(java.awt.Graphics2D g2) {
      ...
   }

   public boolean contains(double x, double y) {
      ...
   }
}
```

The constructor in this example, as required, contains only a `super()` call which passes on the initial attribute values to the `Text` class, while the `draw()` and `contains()` methods of the `DrawableToken` interface are also present. Note that `TokenCanvas`, `InterpretedTokenCanvas` and `DrawableToken` are all found in the `au.edu.monash.csse.tonyj.cider.canvas` package.

While the Drawable Class attribute is most commonly used for `terminal` symbols, it can also be associated with `non-terminal` symbols in the grammar (see Appendix C for more information). It is also possible to define a `terminal` symbol without a Drawable Class, however this is unusual and the Cider Compiler will issue a warning if such a symbol definition is found.

5.3.1 Symbol Attributes

On the second line of the symbol definition example is the list of attributes associated with the symbol being defined. Each attribute is defined by its name, followed by a colon (`:`), followed by its data type. Attributes appear in a comma separated list which can contain any number of (zero or more) attributes.

Each symbol attribute must be given a proper name that is unique to that symbol definition. When the Cider toolkit produces the class for each symbol type, two access methods will be created for each symbol attribute called `set_`*attribute_name*`()` and `get_`*attribute_name*`()` (although depending on the production rules, the `set_`*attribute_name*`()` method will not always be publicly available as described in Appendix B).

As the Cider toolkit must implement each attribute with a Java datatype, this must also be specified. This datatype value can be one of the following.

- Any of the eight primitive data types in Java (`int`, `short`, `long`, `float`, `double`, `char`, `byte` or `boolean`).
- A fully qualified name of any Java class. It is recommended that only Java classes that create immutable objects are used. This is to ensure that attribute values can only be modified via a call to the `set_`*attribute_name* method; otherwise, parse forest inconsistencies may arise at runtime as described below in Section 5.3.2.

- A multi-way constraint variable, specified by the command `@cv`. Such a variable may then be used in any solver constraints specified later in the grammar file, and will otherwise be treated as a variable of type `double` (see Section 1.4 for more information on the multi-way constraint solver used by CIDER). The stay and edit weights of the variable can be specified after the `@cv` string as two (comma separated) positive floating point values inside a pair of square brackets. If these weights are not specified, default weights are used. Details on the stay and edit weights are discussed in Section 5.3.3.
- The name of any symbol type defined in the grammar.

5.3.2 Guidelines if using Mutable Objects as Attribute Datatypes

It is recommended that only immutable objects are used as attribute datatypes. However, sometimes it will be more convenient to use classes that create mutable objects. This can be done safely, however certain precautions must be taken by the application developer to ensure that CIDER Interpreter can maintain parse forest consistency. To illustrate the potential problems with mutable objects, let us look at an example that may lead to an inconsistent parse forest.

Consider the case where there is a symbol with an attribute called `list`, whose datatype is an instance of `java.util.List`. If an instance of this symbol is referred to in the following Java code by the variable `sym`, then the following is possible.

```
List tempList;
tempList = sym.get_list();
tempList.add("New String");
```

Here we have acquired the symbol's `list` attribute (using the `get_`*attribute_name*`()` method), and then added a new string object to it. However, the CIDER Interpreter will be unaware of this change, which may lead to the parse forest being in an inconsistent state. For example, there may production rules whose conditions depend on the contents of the `list`. Changing the contents of the `list` object may invalidate some production rules or make others valid when they previously were not.

To ensure that an inconsistent state does not occur, the `set_`*attribute_name*`()` method must be used after the modification so that the CIDER Interpreter will be aware of the change. This means that previous example needs to have an additional line of code.

```
List tempList;
tempList = sym.get_list();
tempList.add("New String");
sym.set_list(tempList);
```

Now the parse forest should remain in a consistent state.

Note that some attributes may not have a publicly accessible `set_`*attribute_name*`()` method (see Appendix B for details on why this is so). If this is the case, then the attribute should never be modified in any way by the application code – only the CIDER Interpreter should be allowed to modify such attributes.

5.3.3 Setting Stay and Edit Weights for Constraint Variables

Users cannot modify constraint variables directly. Instead, they can only suggest a new value for the variable. The multi-way constraint solver (QOCA) will then check if this new value can be allocated or not, and this may also involve changing the values of other constraint variables depending on the constraints in the solver. The stay weight indicates the importance of not changing a variable if a new value for it has not been suggested (the bigger the stay weight of a variable, the less likely to be modified it is). The edit weight indicates the importance of changing a variable when a new value for it has been suggested (where a bigger edit weight means the variable is more likely to meet its new suggested value).

If the stay and edit weights are not specified, default weights are used (10 for the stay weight, 1000 for the edit weight). Typically one will want the edit weight to be larger than the stay weight used for most variables, otherwise it might never be possible to change the values of any constraint variables that are part of a constraint. Consider the simple constraint $x == y$, where both x and y equal 3.0. If a new value of 7.0 is suggested for x, it will only be possible for x (and y) to be allocated this new value if the edit weight of x is greater than the stay weight of y.

5.3.4 Abstract Symbol Types

If two or more symbol types have many attributes in common, it may be preferable to create an abstract symbol type containing those attributes, and allowing the original symbol types to make use of the abstract symbol type. For example, in the grammar for Finite State Automata, there are two non-terminal symbols called `FinalState` and `NormalState` which use the exact same attributes. Rather than duplicating these attributes in the two symbol definitions, a single `abstract` symbol was created.

```
State (abstract)
   radius:double, mid_x:double, mid_y:double, label:java.lang.String;
```

Given this definition, it is possible for the symbol `FinalState` and `NormalState` to use the abstract `State` symbol in the following way.

```
FinalState (non-terminal) : State;
NormalState (non-terminal) : State;
```

This notion of abstract symbol types is analogous to inheritance in object-oriented programming. `FinalState` and `NormalState` can be thought of as inheriting the attributes of the abstract `State` symbol. It is also possible to have multiple levels of abstraction; that is, it is possible for the `State` symbol to inherit attributes from another abstract symbol. If multiple levels of abstraction are being used, care must be taken to ensure that no dependency cycles occur. Note that in the above example, `FinalState` and `NormalState` do not have attribute lists because they require no additional attributes. If either of them did require additional attributes, these would simply be specified as has been shown in previous examples.

There is also a further benefit in using abstract symbol types. While it is not possible to directly create a `State` symbol, nonetheless, when either a `FinalState` or `NormalState` is created, they can be manipulated as though they were simply instances of `State`. This allows a useful level of generalization when defining production rules. For example, when defining the production rule for a transition between two states in a Finite State Automaton, there are four possible combinations for the source state and destination state (final-final, final-normal, normal-final and normal-normal). This would normally require four separate production rules to be written. However, by making use of the abstract symbol `State`, only one production rule is needed as a `State` can be considered independently of whether it is actually implemented as a `FinalState` or `NormalState`.

To recap, the symbol definitions consist of the following components in the following order:

- the name of the symbol,
- parentheses containing the type of symbol (`terminal`, `non-terminal`, `start` or `abstract`),
- square brackets containing the class used to draw the symbol (usually only for `terminal` symbols),
- a `:` character followed by the name of the symbol that it inherits from (only if it inherits from another symbol),
- a comma separated list of attributes (written as *name*: *datatype*) associated with this symbol, and
- a `;` character to conclude the definition.

5.4 Specifying the Production Rules

Production rules are used to define the hierarchical relationships between various symbols in the grammar. When a production rule is successfully applied, new symbols are added to the parse forest (along with their relationship to existing symbols in the parse forest). Production rules require the specification of four components: the input symbol(s), the output symbol(s), the conditions that the input symbol attributes must meet in order for the production rule to apply, and information on how to set the attribute values of any newly created output symbols.

The production rule definitions are specified in the grammar file after the symbol definitions, using the `production` command. Using this command, any number of production rules (zero or more) can be specified. Productions can either be named, or they can be anonymous. An example of declaring a production called `normalStateProduction` is the following.

```
normalStateProduction:production { ... }
```

The format requires the name of the production to be specified first (which must be a proper name), followed by the `:` character, followed by the keyword `production` (where the contents of this command must appear between the `{` and `}` characters). Anonymous productions are almost identical, except that no name is specified.

```
:production { ... }
```

Thus, the format for anonymous productions begins with the `:` character. Note that productions only need to be named if they will be referred to explicitly inside transformations (see Section 6).

Inside the `production` command, the first things to be specified are the output and input symbols. Each symbol is specified by giving it a proper name, following this name with a `:` character, and then following this character with the name of a particular type of symbol. For example, one of the input symbols required to create a `NormalState` is a `Circle`. Thus, to specify that our production rule involves a `Circle` called `c`, we include `c:Circle`. Note that each symbol name must be unique in the production in which it is specified.

All of the input symbols should form a comma separated list, as should all of the output symbols. There must be at least one input symbol and at least one output symbol (but as the grammar used is an unrestricted grammar, there is no upper limit on the number of input or output symbols). The input symbol and output symbol lists are separated by the `::=` string, with the output symbols placed on the left of this string, and the input symbols placed on the right of this string. For example, for the production to create a `NormalState` from a `Text` and `Circle` symbol, we have the following.

```
n:NormalState ::= t:Text, c:Circle
```

Note that the output symbols must be of type `non-terminal` or `start`, whereas the input symbols can be of any type (except they shouldn't be `start` symbols). This means that input symbols can be of type `abstract`, which allows for more generalization when specifying a production rule (for example, by specifying `s:State` and an input symbol in a production rule, the rule can apply with `s` being either a `NormalState` or a `FinalState`).

It is also possible for a production rule in CIDER to receive a collection of input symbols of a specific type. This is done by adding the `*` character to the end of an input symbol specification, forming an input *symbol collection*. For example, if the input symbol list included `cs:Circle*` as one of the parameters, it would mean that `cs` referred to the set of all unreduced `Circle` symbols in the parse forest. There must be at least one such symbol present for the production rule to apply, and none of the other input or output symbols can be of the same type (that is, `Circle`). At the moment, CIDER does not allow any constraints to be put on input symbol sets, so it would not be valid for `cs` to be referred to in the constraint set of a production rule (because it refers to a collection of symbols, rather than a single symbol). Note that if such a production rule is successfully applied, and later one of the `Circle`s is removed from the parse forest or a new one is added, the production is invalidated and will have to be reapplied (although this will happen automatically).

Note: Productions are evaluated in the order that they are written in the grammar specification. Depending on the use of negative context in a grammar, this ordering may be significant.

5.4.1 The Constraint Set

After the input and output symbols of a production have been defined, the constraint set needs to be specified. The constraint set is the set of conditions that must be met before the production can be applied. Not only must these conditions be met during the evaluation of the production, but the conditions must continually be met after the production has been successfully applied as well. If, for example, at a later stage an attribute value of an input symbol is changed so that one or more conditions are no longer met, the production becomes invalidated and any output symbols of the production are removed from the parse forest.

The constraint set is specified using the `where` keyword. If no conditions need to be specified, then the constraint set is declared in the following way.

```
where true;
```

If conditions do need to be specified, then the declaration is of the form

```
where { ... }
```

where the individual conditions are specified between the `{` and `}` characters. Any number of conditions can appear inside a `where` definition, with these conditions forming a conjunctive set (that is, for the constraint set to be true, every individual condition inside it must be met).

Most individual conditions will be simple comparisons of the form

Argument1 Relationship Argument2;

5.4.2 Arguments

The arguments in a simple comparison can be one of four possibilities: an attribute value, a symbol value, a constant, or the value returned by a static Java function. Attribute values are of the form

symbol_name.*attribute_name*

For example, if a production includes a `Circle` called `c` as an input symbol, and this symbol has an attribute called `radius`, then the attribute value is referred to by the string `c.radius`. Similarly, the actual circle symbol itself (`c` in this case) can also be used as an argument that represents a symbol value.

Constant values can be any string (specified inside double quote marks, `"`), any character (specified inside single quote marks, `'`), any valid numerical value (which may optionally be preceded by a negative sign), or any of the three Java keywords, `true`, `false` or `null`. Also, snippets of literal Java code can be placed inside a `@java` command. The `@java` command is followed by a pair of parentheses that contains a quoted string containing some Java code. The following is an example of the `@java` command.

```
@java("2.0 * Math.PI")
```

This command returns the value 2π, as `PI` is a static constant defined in the `Math` class in the `java.lang` package. Note that code that appears inside a `@java` command must represent a value and not a code statement (that is, there should be no `;` character present).

The values returned by static Java functions (either part of the core API or user-defined) can also be used as arguments. They are specified using the following format

fully_qualified_classname.*function_name*(*Argument1*, *Argument2*, ...)

where the number of arguments given must match the number of arguments of the specified function. Each argument of the function must be either an attribute value, symbol value, constant, or the value returned by a static Java function (yes, this means that a function call can be specified from within the argument list of another function call, with no limit on the degree of nesting). Each argument must also match the datatype that the function is expecting for that argument. The function must be static, and it cannot be a function that throws exceptions.

As an example of a function call, consider again the `Math` class which has a `static` function `sqrt` that takes a single argument, and returns its square root. To get the square root of the `radius` of a `Circle` called `c`, we would use the following.

```
java.lang.Math.sqrt(c.radius)
```

5.4.3 Relationships

Aside from arguments, the other component of a simple comparison condition is the comparison relationship. The sort of comparison that can be used will depend on the datatypes of the arguments being compared. For most primitive datatypes (such as `int`, `double`, etc.) the following six relationships are possible: `==`, `!=`, `<`, `>`, `<=` and `>=`. The meaning of these six relationships is the same as when they are using in the Java language. However, they cannot be used with objects in Java. Fortunately, Java also makes it possible to check if two Java objects (such as strings) are equal or not. Thus, CIDER also includes the relationship `eq` to represent object equality, and `ne` to indicate that two objects should not be equal. (Note that these two possibilities make use of the `equals()` method in the Java `Object` class (which can be overridden by derived classes), and thus will determine object equality only as well as this method's implementation.)

Given these definitions of arguments and comparison relationships, the following are all valid examples of the most typical type of condition that appears in a constraint set. (Note that they all end in with a `;` character.)

```
t.label eq "Hello World";
java.lang.Math.sqrt(c.radius) > 5.0;
java.lang.Double.isInfinite(c.radius) == true;
c.radius != @java("Double.NaN");
```

5.4.4 Multi-way Solver Constraints

CIDER uses a multi-way constraint solver called QOCA (see Section 1.4), which operates over the domain of real numbers and supports linear constraints involving both equalities and inequalities.

Solver constraints that will be enforced by QOCA are specified using the `@sc` command. The `@sc` command takes two arguments, which appear (comma separated) between a pair of parentheses. The first argument is a tolerance amount that applies to the constraint. This must be a valid real number. Consider the constraint $x == 7.0$. If we use a tolerance value of 2.0, then this constraint will be satisfied if x has a value between 5.0 and 9.0 inclusive. To require x to have a value of exactly 7.0, a tolerance value of zero must be used. Note that the tolerance amount is only used in the initial evaluation of the constraint. Thus, while an x value of 6.0 will satisfy the constraint $x == 7.0$ with a tolerance of 2.0, the moment the constraint is passed to the constraint solver (which will occur if a production rule is successfully evaluated), the x value will be adjusted to equal exactly 7.0.

The second argument that the `@sc` command takes is the constraint string. This is a quoted string that defines the constraint using the following format.

- The format for defining a constraint variable is `[`*symbol_name*`.`*attribute_name*`]`
- Polynomial terms are specified by a coefficient followed by a constraint variable, as in the following two examples (with no `+` symbol allowed on positive coefficients):

  ```
  5.0 [c.radius]          -4.3 [c.radius]
  ```

- If the coefficient is `1.0`, it may be omitted from the polynomial term.
- If the coefficient is `-1.0`, it may be represented simply by a `-` symbol.
- Polynomials are specified by linking polynomial terms with `+` symbols:

  ```
  5.0 [c.radius] + [t.mid_x] + -2.0 [t.mid_y]
  ```

- Constants are just real numbers like coefficients (with no + symbol allowed on positive constants).
- Valid relationship strings are ==, <= and >= (as used in most programming languages).
- A constraint is defined as a polynomial, linked by a relationship string to a constant, for example:

```
-1.0 [t.mid_x] + [t.mid_y] >= 10.0
```

Note: Subtraction between attributes must be written in the form `[t.mid_x] + -[c.mid_x]`. The intuitive `[t.mid_x] - [c.mid_x]` approach will be parsed by the Cider compiler but will not be evaluated as expected. Consequently, any production or transformation which uses the incorrect syntax may not be applied as expected.

The constraint string should only refer to attributes which have been specified as constraint variables using the `@cv` command in its symbol definition. Examples of the `@sc` command are the following.

```
@sc(0.0, "[c.radius] >= 7.0");
@sc(2.0, "[t.mid_x] + -[c.mid_x] == 0.0");
```

The `@sc` command is always concluded with a `;` character.

5.4.5 Existential Symbols

When a production rule is successfully applied its input symbols are reduced, meaning that they cannot be used as input symbols in another production rule. However, it is often desirable for a production rule to consider symbols without those symbols being reduced. For example, for each FSA transition, there should be a state where it starts and a state where it ends. However, if these states were reduced every time a transition was formed, then there could only ever be one transition per state, which is clearly an unworkable restriction. To get around this, constraint sets can also refer to existential symbols.

Existential symbols are symbols that exist in the parse forest and can be referred to in a constraint set, but are themselves unaffected by a successful application of a production. If an existential symbol was not reduced before the production was applied, it will not be reduced as a result of the application. Similarly, existential symbols that were reduced before the production was applied will remain reduced. Also, if the existential symbol was not reduced when a production was applied, but is later reduced by another production (or vice-versa), this will have no impact on the original production for which the symbol was an existential symbol. Existential symbols may be of any type (`terminal`, `non-terminal`, `abstract` or `start`).

Existential symbols represent conditions on a production, and are therefore specified in the constraint set, using the `exists` command. The format for this command is the `exists` keyword, followed by a symbol specification (with a unique symbol name), followed by a constraint set within which the existential symbol can be referred (it cannot be referred to outside this constraint set). The following is an example.

```
exists s:State
where { ... }
```

The `exists` condition in this example will be met if a specific `State` symbol can be found in the parse forest for which the conditions in the `where` clause of the `exists` command are met. (Note that the symbol found cannot be an input symbol of the production, or used by another `exists` condition.) In such a scenario, `s` will refer to a specific symbol in the parse forest. While only one existential symbol can be defined for each `exists` command, the constraint set associated with that command can also contain other `exists` commands, and thus unlimited nesting is possible.

The above example describes how to specify a positive existential relationship. It is also possible to define negative existential relationships. This can be specified using the command `not_exists` instead of `exists` (the format and workings of the command is otherwise identical). The `not_exists` condition will only return true if there does *not* exist a symbol which meets the conditions specified within its constraint set. Again, there is no restriction on nesting, with it being possible to define an `exists` command within the constraint set of a `not_exists` command, and vice versa.

Note that when a `not_exists` command is used, every symbol in the grammar (of the type specified in the `not_exists` command) must be checked to ensure this condition is met (with the exception of any input symbols or existential symbols of the same type within the production). Thus, the `not_exists` condition in the following example will only be true if there are no `Circle` symbols for which the conditions in the `where` clause of the `not_exists` command are met.

```
not_exists c:Circle
where { ... }
```

A consequence of this is that, unlike the case with positive existential symbols, the symbol `c` in the above example will not refer to a particular symbol in the parse forest if the condition is met. This allows us to introduce the notion of a *referable* existential symbol.

5.4.6 Referable Existential Symbols

In this section we specify exactly what a *referable* existential symbol is. An existential symbol is one defined in either an `exists` or a `not_exists` condition. A *referable* existential symbol is one where, if the overall production condition holds, then it refers to a single specific symbol in the parse forest. Let's look at an example constraint set.

```
where {
   exists c:Circle
   where {
      exists s:Square
      where {
         c.colour == s.colour;
      }
   }
}
```

This example is equivalent to saying *this condition holds if there is a circle for which there is also a square with the same colour.* If the parse forest contains a one red circle and one red square, the condition will hold. Furthermore, `c` and `s` will refer to those specific symbols in the parse forest, and thus they will be considered *referable.* What if there were a red and blue circle and a red and blue square? In this case, the condition would still be satisfied as soon as a valid pair of symbols for `c` and `s` is found. It could be the red pair of symbols or the blue pair. All that matters is that once symbols have been found that satisfy the condition, then `c` and `s` will refer to those particular symbols from then on. This contrasts with the following example.

```
where {
   not_exists c:Circle
   where {
      not_exists s:Square
      where {
         c.colour == s.colour;
      }
   }
}
```

This example is equivalent to the sentence *this condition holds as long as there is not a circle for which there is not a square of the same colour.* In other words, for `all` circles there must be present a square with the same colour. Consider the situation where there are two circles in the parse forest, one red and one blue. The condition will hold as long as there is at least one red square present and one blue square present. (Note that there may be multiple red or blue squares present, or squares of other colours. Also, if there were two red circles, there would still only need to be one red square.) Importantly, if the condition holds, `c` and `s` will not refer to specific symbols in the parse forest; `c` will be used to check *all* circle symbols in the forest, and while s will be used to represent a specific square symbol for each particular circle, different circles may require different squares. Thus, c and s do not refer to a single specific symbol in the parse forest and so are not classified as *referable* existential symbols. The same is true of the following example, which is only slightly different (with an exists condition for the square s).

```
where {
   not_exists c:Circle
   where {
      exists s:Square
      where {
         c.colour == s.colour;
      }
   }
}
```

If we again have a single red and blue circle in the parse forest, the condition will only hold if there are no red and blue squares also present. However, there can be any number of squares with different colours. Thus in this example also, c and s represent existential symbols that do not correspond to a single specific symbol in the parse forest, and so are not *referable*.

If we consider the main where clause that encompasses all other conditions as the *base* where clause, then it is easy to determine which existential symbols are *referable* and which are not. Any exists symbol immediately within the base where clause is a referable existential symbol. Also, any exists symbol immediately within the where clause of another referable existential symbol, is also a referable existential symbol itself. All not_exists symbols are not referable existentials. Also, any existential symbol declared inside the where clause of a not_exists symbol (using either exists or not_exists), whether declared immediately or nested further inside other existential declarations, is also not a referable existential symbol.

5.4.7 Initializing and Setting Attribute Values

When a production rule is applied, new non-terminals are added to the parse forest. These output symbols will have attributes whose values need to be set, and this attribute setting follows the constraint set.

There are actually two types of attribute setting. Attribute values can be either given an initial value which may be changed later on, or they can be set to always equal a value that is dependent on the values of other attributes (belonging to other symbols in the production rule). The former is specified inside the init command, and the latter is specified inside the set command. The latter case is achieved through the use of a one-way constraint solver that is built into the CIDER toolkit. Every time an attribute value is changed, CIDER checks to see if any other attribute values depend on this value (as specified within the set command). If this is the case, these attribute values are updated accordingly. Note that one-way constraints have been widely used in GUIs and can be extremely efficiently solved [2].

The first command that follows the constraint set is the init command. The contents of this command appear between the { and } characters. Within this command are any number of lines of the form

symbol_name.*attribute_name* := *Argument*;

This line begins by specifying a particular attribute, for example, c.radius (in this case c would have to be an output symbol of the production rule which has an attribute called radius). Then the string := must appear, followed by an argument (as defined above in Section 5.4.2). The argument should correspond to a value of the same datatype as the attribute being initialized. Each line should also finish with the ; character. Note that inside an init command, if any argument refers to another attribute value, only its current value will be considered. The argument may refer to attributes that belong to any of the input symbols or referable existential symbols in the production rule. If no attribute values need to be initialized in this way, then the init command can be left out altogether.

Following the `init` command is the `set` command. This command works in a similar way as the `init` command, however, any attribute settings that involve other attribute values are passed to the one-way constraint solver to be maintained after the production rule has been applied. There are, however, two important options within the `set` command.

The first option is that the role of the one-way constraint solver may be replaced by the multi-way constraint solver for any attribute setting involving constraint variables. This is done by specifying that the argument of a particular attribute setting is a constraint expression, by using the command `@ce`. This command takes one argument (inside a pair of parentheses), which is a quoted string containing a constraint expression. The format of the constraint expression is based on that used for specifying solver constraints (see Section 5.4.4) with the following requirements.

- A constraint expression is defined as a polynomial linked by a `+` symbol to a constant, for example:

```
[t.mid_x] + -2.0 [t.mid_y] + -6.5
```

- A constraint expression may also be defined as a constant by itself, or a polynomial by itself.

Consider the following example of an attribute setting involving the one-way constraint

```
c.radius := t.length;
```

In this case, every time the attribute referred to by `t.length` is changed, the `c.radius` attribute is updated to equal the same value. However, the reverse is not true. In fact, to prevent the parse forest from becoming inconsistent, Cider will not allow the attribute `c.radius` to be independently modified. Now, consider the case where a multi-way constraint solver is used.

```
c.radius := @ce("[t.length]");
```

This effectively defines the multi-way constraint `c.radius == t.length`. In this case, if either `c.radius` or `t.length` is modified, the other attribute will be updated to the same value by the multi-way constraint solver. This provides a far more flexible structure preserving mechanism.

In short, if a constraint variable is updated using the one-way constraint solver, this can limit the values that can be assigned to it by the multi-way constraint solver which will result in a less flexible specification. See Appendix D for more detail on what happens in such a scenario (where the one-way and multi-way constraint solvers are forced to work together to maintain parse forest consistency).

The second option is that inside the `set` command, the attributes of input symbols may also be specified, allowing attribute updating to flow not only up a parse tree, but also down if required. Before allowing such settings, Cider checks to make sure that the setting will not result in a conflict (which can happen, for example, when a symbol is the output symbol of one production and the input symbol of another, and both productions attempt to set one of its attributes). For a complete explanation of what attribute settings Cider permits, see Appendix B.

There is one further point to note with the `set` command. It is valid to specify an attribute value to equal a symbol in the parse forest (either an input symbol or a referable existential symbol, provided that the attribute's datatype was specified accordingly). However, if the symbol in question later has an attribute modified or is even removed from the parse forest, no updating will occur to the attribute setting. This could lead to unexpected behaviour. Therefore, caution should be exercised when setting attribute values to equal symbols in the parse forest. This holds both when an attribute is set using a symbol value directly, or is set using a function value where one or more arguments of the function are symbol values.

The `set` command is the last command that appears inside a `production`. As with the `init` command, if no attribute values need to be set, then the `set` command can be omitted. However, note that all of the output symbol attributes must be accounted for exactly once in either the `init` command or the `set` command.

6 Format for the Transformations File

For any diagrammatic language for which an editor is to be created, it may also be desirable to specify certain transformations. Transformations are essentially mechanisms for converting part of a diagram parse tree from one valid structure into a different valid structure. All transformations must be specified within a single text file, and this file must be separate from the file containing the grammar. This section describes the format for the transformations file, which can contain any number of (zero or more) transformation specifications.

6.1 Basic Transformations

Basic transformation rules are similar in many respects to production rules. Like production rules, they have input symbols that must met a set of conditions before the rule can apply. The primary difference between the two is that unlike production rules, transformations only need to be considered at the time they are applied, whereas production rules can still become invalidated long after they are applied. Other key differences are that the input symbols of a transformation rule are removed from the parse forest when it is applied, and transformations can involve a more complex sequence of actions when they are applied (rather than just the creation of non-terminals).

Transformation rule definitions are specified in the transformations file using the `transformation` command. Using this command, any number of transformation rules (zero or more) can be specified. Each transformation must have a unique name. An example of declaring a transformation called `TwoStateProcessing` is the following.

```
TwoStateProcessing:transformation { ... }
```

The format requires the name of the transformation to be specified first (which must be a proper name), followed by the `:` character, followed by the keyword `transformation` (where the contents of this command must appear between the `{` and `}` characters).

Inside the transformation, the first thing to be specified is the input symbols. Note that this is unlike production rules where output symbols must be specified first. Each input symbol is specified by giving it a proper name, following this name with a `:` character, and then following this character with the name of a particular type of symbol (the same as for production rules). Unlike production rules, no symbol collections can be used with transformations. Also unlike productions, it is possible to specify no input symbols. If no input symbols are required, then the constraint set will be the first thing specified inside the `transformation` command.

Input symbols can be of any symbol type (`terminal`, `non-terminal`, `abstract` or `start`). When a transformation is applied, all input symbols will be removed (in the order they are listed). If an input symbol is non-terminal, not only will it be removed, but its entire sub-tree in the parse forest will also be removed. So if a `NormalState` input symbol is removed, the `Circle` and `Text` symbol that makes up the `NormalState` symbol will also be removed from the parse forest. Naturally, if the symbol specified is also used as an input or referable existential symbol in a production, that production will be invalidated and any output symbols generated by it will also be removed.

Following the input symbols inside the `transformation` command is the constraint set. The constraint set format is the same as that for production rules (see Section 5.4.1), with the exception that solver constraints (specified using the `@sc` command) cannot be included. This is because the conditions of a transformation should only be considered at the time it is applied, and not enforced afterwards.

6.1.1 Action List

If the conditions in a transformation's constraint set are met, then the actions associated with the transformation can be applied. (Note that these actions take place *after* the input symbols have been removed from the parse forest, although the values of the attributes associated with the input symbols can still be accessed.) The action list is specified inside the `apply` command which follows the constraint set. (If no actions are needed, this command may be omitted.)

```
apply { ... }
```

Inside this command, three types of action can be specified. These actions are: the creation of a new terminal symbol `(nts)`, the application of a specific production rule `(ap)`, and the modification of an attribute value `(set)`. Note that the transformation will fail unless all of the actions in its action list can be successfully applied in the order they are listed.

The creation of a new terminal symbol is specified in the following way.

`(nts)` *symbol_name*`:`*symbol_type*`(`*Argument1*`,` *Argument2*`, ...);`

For this command, the symbol name must be unique, the symbol type must be a type of terminal symbol, and initial values for all of the attributes of the terminal symbol must follow in a comma separated list between a pair of parentheses. The arguments for this list are the same as those defined for production rules (see Section 5.4.2). The arguments can refer to the attributes of any symbol defined up to that point in the transformation rule. The order of the arguments corresponds to the order in which the attributes are defined in the symbol definition. If the symbol extends an abstract symbol, its local attributes are specified first, and then the attributes of the abstract symbol follow. An example of this command is the following (where a symbol `c` has already been defined).

```
(nts)  t:Text("New Text", c.mid_x, c.mid_y);
```

This symbol, `t`, can now be referred to in subsequent actions specified in the action list.

The application of a production is specified by the command `(ap)` followed by the name of the production rule to be applied. Thus, anonymous productions cannot be referred to in a transformation. The name of the production rule can optionally be followed by the command `using(...)`, which allows mapping to input symbols in the production, and from output symbols in the production. For example, consider the production rule `normalStateProduction`, with the output symbol `n:NormalState` and the input symbols `t:Text` and `c:Circle`. If a `Text` symbol called `t1` has been defined in the transformation rule, and a `Circle` symbol called `c1` has also been defined, then the following example can be specified.

```
(ap)  normalStateProduction using(t1 -> t, c1 -> c, n1 <- n);
```

Note that the string `->` is used to specify an input mapping, and `<-` is used to specify an output mapping. In this example, the production is applied only considering `t1` and `c1` as possible candidates for input symbols. If successful, the output symbol produced by the production will be mapped to the name `n1`. Using an output mapping allows the output symbol to be referred to in subsequent actions in the action list. Note that each input and output mapping is optional. If, for example, the mapping `t1 -> t` wasn't given, then the production would simply search the parse forest for all possible candidate `Text` symbols.

A modification of a symbol attribute value is specified in the following way

`(set)` *symbol_name*`.`*attribute_name* `:=` *Argument*;

Thus, aside from the preceding `(set)` string, the format for this command is the same as that used to initialize an attribute in a production rule using the `init` command (see Section 5.4.7).

All action specifications in the action list must be concluded with the `;` character. The action list is the last element of a basic transformation.

Note that for a basic transformation to be successfully applied, not only must the conditions in the constraint set be met, but it must also be possible to successfully apply all of its actions in the order they are listed. If any actions cannot be applied, the transformation itself will be deemed non-applicable, and the state of the parse forest will return to what it was before the attempt to apply the transformation was made.

6.2 Parallel Transformations

When a call is made to apply a basic transformation, there may be many possible symbol combinations that the transformation can apply to. Normally the first symbol combination that is found will be used. However, it is also possible to apply basic transformations in a parallel manner. This means that every possible way in which the transformation could be applied (for a given diagram instance) will be carried out simultaneously. Note that if there is cross-over in the possible symbol combinations, unexpected or undesired results can occur as CIDER will refuse to allow the state of the parse forest to become inconsistent. It is the application programmers responsibility to be aware of this and to deal with such possibilities.

6.3 Transformation Disjunctions

It is possible to specify disjunctions of basic transformations. For example, a two transformation disjunction is essentially the same as saying "apply either Transformation A or Transformation B." Disjunctions can also involve more than two basic transformations. They are specified with the `disjunction` command as follows.

```
ProcessInput:disjunction { ... }
```

Like the `transformation` command, this command begins with a unique proper name, followed by the `:` character, followed by the `disjunction` keyword, with the contents of the disjunction command appearing between the `{` and `}` characters.

The contents of the `disjunction` command is simply a sequence of basic transformation names (two or more) separated by the `|` character, with the sequence concluded with the `;` character. So, if for example, `TwoStateProcessing` and `OneStateProcessing` are basic transformations, a valid content for the `disjunction` command would be as follows.

```
TwoStateProcessing | OneStateProcessing;
```

Disjunctions will only fail to be applied if none of the basic transformations they are composed of can be applied. The basic transformations will be attempted in a left to right order. Note that like basic transformations, it is also possible to apply disjunctions in parallel.

6.4 Compound Transformations

With the CIDER toolkit it is also possible to go beyond disjunctions, and make use of the full power of regular expressions. This allows the use of disjunction, concatenation, and repetition (not to be confused with parallel application of a transformation). These operations can be used within the `compound_transformation` command, the format of which is the same as was used for the `disjunction` command in the previous section; that is, a unique proper name, followed by a `:` character, followed by the `compound_transformation` keyword, with the contents of the command being placed between the `{` and `}` characters.

Inside the command is a single regular expression. It consists of the names of basic transformations (or disjunctions), with the `|` character used to represent disjunctions, the `*` character used to represent zero or more repetitions, and adjacency to represent concatenation. Parentheses can also be used as required. Order of precedence is that normally used in defining regular expressions. Thus, if `A`, `B`, `C` and `D` are all basic transformations, the following is a valid regular expression which can be used inside a compound transformation.

```
( A | B * ) C! D;
```

Note that in the above example, the basic transformation `C` has the character `!` appended to it. This means transformation `C` should be applied in parallel. While basic transformations within a compound transformation can be applied in parallel, the compound transformation itself cannot be applied in parallel.

A compound transformation will fail if the state of the parse forest will not allow basic transformations to be applied in a sequence that will fulfill the requirements of the regular expression. Consider again the above example. If `C` can be applied (in parallel) but `A`, `B` and `D` cannot be applied then the following will happen. `A` will be attempted and fail, so the disjunction will instead apply `B*`. Even though `B` cannot be applied, `B*` will succeed because zero repetitions is still valid. Thus the disjunction will be successful. Then `C` will be applied in parallel and succeed. Finally, to complete the concatenation, `D` will be attempted but will fail, resulting in the overall failure of the compound transformation, and any basic transformations that had already been carried out as part of the compound transformation will be undone.

6.5 Special Pre-defined Transformations

There are two special pre-defined transformations in the CIDER toolkit: they are `CheckForest` and `ForceDirectedLayout`. These special transformations are designed to be included in compound transformations, where they each perform a specific task. Naturally, no other transformations that are defined are allowed to use the same name as these two transformations.

The `CheckForest` transformation is provided to automatically bring the parse forest up date, and therefore simplify the action lists of basic transformations. More specifically, consider a basic transformation in which the action list requires a new sub-tree to be added to the parse forest. The new terminal symbols are all added using the `(nts)` command. However, there may then be many productions that need to be applied, and specifying them individually using the `(ap)` command would be tedious and error prone. Indeed, in some circumstances it may be difficult to know all of the productions that need to be applied.

To avoid having to specify all of the productions required, the solution is to only create the terminal symbols required in the basic transformation. Then, the basic transformation can be used in a compound transformation, concatenated with the `CheckForest` transformation. If the basic transformation is called `CreateSubTreeTerminals`, then the compound transformation can be specified as follows.

```
CreateSubTree:compound_transformation {
    CreateSubTreeTerminals CheckForest;
}
```

After the `CreateSubTreeTerminals` transformation has been applied, the special `CheckForest` transformation will be applied. This transformation will check the parse forest and evaluate all production rules, and thus bring the parse forest up to date by applying any productions that can be applied (and also invalidating any productions that need to be invalidated).

Note that the `CheckForest` transformation will always report successful execution, even if no changes to the parse forest had to be made (this ensures that the `CheckForest` transformation will never hinder the successful application of any compound transformation it appears in). Of course, using the `CheckForest` transformation will not allow for specific symbol mappings to be used. However, in most situations, the optional symbol mapping of the `(ap)` command is not necessary anyway. (While the `(ap)` command may provide some performance benefits, the difference is not likely to be significant.)

The other special transformation is `ForceDirectedLayout`. If the grammar has been set up to include components for force directed layout (see Section 7 for details), this allows the animation of the layout to be specified within a compound transformation. Then, whenever that compound transformation is called, the force directed layout animation will be invoked at the point where it occurs in the regular expression. This animation will incrementally layout the diagram in the `TokenCanvas`, according to information provided by the `GraphNode` and `GraphEdge` symbols. As with the `CheckForest` transformation, the `ForceDirectedLayout` transformation will also always report successful execution, even if nothing needed to be done.

One point to note is that the force directed layout animation may occur, and then be suddenly invalidated if there are further elements in the compound transformation whose application may not be successful. For example, consider the following regular expression.

```
TransformationA ForceDirectedLayout TransformationB;
```

If `TransformationA` is successfully applied, then the `ForceDirectedLayout` transformation will occur, animating the layout. However if following this `TransformationB` is unsuccessful, all actions of the compound transformation will be undone. Since it would be disconcerting for a user to see an animation change the layout of a diagram, only for the new layout to be instantly undone, it is recommended that the `ForceDirectedLayout` transformation is only used as the last element in the regular expression of a compound transformation.

6.6 Programmatic Transformations

CIDER's transformation mechanism is not the only way in which transformations can be applied. It is also possible to define transformations in the application's Java code. While for many types of transformation it will be simpler to use the CIDER transformations, there are also situations for which the CIDER transformations are not well suited (for example, when set manipulation is involved and set membership is dependent on complex conditions). In these situations, programmatic transformations may be more appropriate.

A programmatic transformation is essentially just a manipulation of the diagram that is specified in the Java code of the application. Technically, whenever a new terminal symbol is added to a diagram (by using the `add()` method in the `InterpretedTokenCanvas` class), a programmatic transformation has taken place. This is because the diagram has changed (a new symbol is present) as the result of the execution of Java code (where the code did not refer to a CIDER transformation). The same is true when a terminal symbol is removed, or when an attribute of a symbol is modified.

Just as a basic CIDER transformation has conditions, so too a programmatic transformation can also have conditions. These conditions are specified by considering the structure of the parse forest and the values of symbol attributes via programmatic means. For example, each symbol in the grammar is implemented by a class that inherits the class `GrammarSymbol`. Thus, information on the symbol's place in the diagram structure can be accessed via methods such as `getParents()` and `getChildren()`. Of course, the `get_`*attribute_name*`()` methods allow attribute values to be accessed. The structure of the parse forest can also be monitored as it is being built by using the `GrammarSymbolListener` interface. (Both `GrammarSymbol` and `GrammarSymbolListener` are in the package `au.edu.monash.csse.tonyj.cider.interpreter`; see the Javadocs for more information.)

As for actions to manipulate the diagram, it is possible to add and remove terminal symbols (via the `add()` and `remove()` methods in class `InterpretedTokenCanvas` or the `addTerminalSymbol()` and `removeTerminalSymbol()` methods in class `Interpreter`), and for attribute values to be modified via the `set_`*attribute_name*`()` methods (see Appendix B for details on which attributes can be modified). While it is not possible to programmatically call a specific production to be applied, for a well-defined grammar this should not be an issue as CIDER will process each diagram manipulation automatically. If the application developer wants to make several changes at once and only update the parse forest at the end, automatic parse forest checking can be temporarily turned off using the `setAutoForestChecking()` method in the `Interpreter` class.

7 Force Directed Graph Layout

The CIDER toolkit has the ability to allow for force directed graph layout of diagrams. This is a convenient feature for diagrams which have a graph-like structure (such as FSA diagrams). Note that this capability as it is currently implemented is rudimentary (for example, it does not support features such as orthogonal node connectors), and no other types of diagram layout have yet been built into the toolkit. However, the component based design of the CIDER toolkit is such that it should be readily extendible, allowing application programmers to add their own form of diagram layout as desired.

The force directed graph layout in CIDER relies on two abstract symbol definitions, `GraphNode` and `GraphEdge`. These symbol definitions are available to be used by any grammar specified for CIDER and thus these names are also reserved names (application developers cannot define their own `GraphNode` or `GraphEdge` symbol definition). The symbol definitions are as follows, where both symbol types have four attributes.

```
GraphNode (abstract)
   mid_x:@cv, mid_y:@cv, width:double, height:double;
GraphEdge (abstract)
   mid_x:@cv, mid_y:@cv, node1:GraphNode, node2:GraphNode;
```

A `GraphNode` is defined by its location in a diagram (specified by the attributes `mid_x` and `mid_y`, both of which are constraint variables). Also, a `GraphNode` does not represent an infinitely small point, but rather has an area which is specified by the `width` and `height` attributes. A `GraphEdge` is defined by the two `GraphNode` symbols that it connects (specified by attributes `node1` and `node2`), and also has a mid-point (specified by the attributes `mid_x` and `mid_y`, both of which are constraint variables).

The presence of a mid-point on the `GraphEdge` symbol is so that curved graph edges can be represented. For layout purposes, this mid-point is considered to be a graph node in the same way that the `GraphNode` symbol is. However, this graph node is simply represented as a point (it has no area). This means that the `GraphEdge` symbol is interpreted as actually representing two graph edges: one from `node1` to the `GraphEdge` mid-point, and one from the mid-point to `node2`. This is illustrated in Figure 8, which shows how the layout of two `GraphNode` symbols connected by a `GraphEdge` symbol is interpreted.

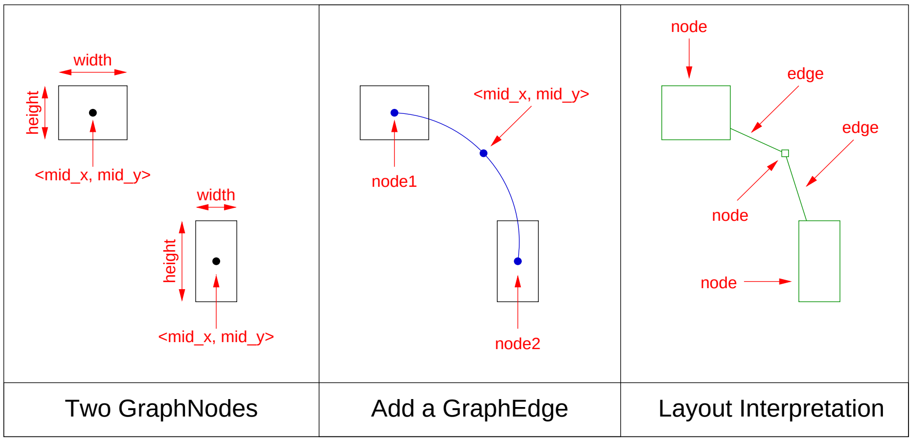


Figure 8: How `GraphNode` and `GraphEdge` symbols are interpreted by CIDER.

The symbols `GraphNode` and `GraphEdge` can be incorporated into any grammar by using them as one would use any `abstract` symbols (see Section 5.3.4). Whenever these symbol types are incorporated into a grammar, then force directed graph layout will automatically be available.

The force directed graph layout algorithm has two primary components. There is a particle repulsion force between each pair of nodes in the graph (whether they are connected by an edge or not). This particle repulsion force decreases quadratically as the distance between the two nodes increases. There is also a spring force between each pair of graph nodes connected by a graph edge. This spring will either pull the nodes together or push them apart depending on whether the distance between the nodes is greater or less than a given equilibrium distance. Parameters for the force directed graph layout can be set in the `Interpreter` class using the `setForceDirectedLayoutParameters()` method (see the Javadocs that come with CIDER for details on using this method). For more information on force directed graph layout algorithms, see [4].

To actually layout a diagram, the method `applyForceDirectedLayout()` in the class `Interpreter` must be called. (It is also possible to layout a diagram from within a compound transformation – see Section 6.5.) The laying out of a diagram takes place incrementally, so this method actually produces an animation of the diagram layout. One benefit of this is that, because each animation step is fairly small, the user is not likely to lose track of where different diagram elements are being relocated to. If the layout simply took place in one step, the user may end up losing all feel for what items have moved where, and may even incorrectly assume that they are looking at a different diagram rather than a different version of the same diagram. The `applyForceDirectedLayout()` method takes one argument, which is the minimum amount of time (in milliseconds) between each animation step. Note that if a diagram is complicated, it may actually take longer than the time specified to perform each animation step.

(Note that at the time of writing, the animated layout does not appear on systems using Mac OS X. The problem appears to be an incorrect implementation of the `paintImmediately()` method of class `javax.swing.JComponent` in Apple's Java distribution.)

8 Debugging Cider Applications

Given that the method for creating a SDE application with Cider is not trivial, it is inevitable that some debugging will be required. This section examines the three main levels where errors can occur during application development, and provides information to assist the debugging process at each level.

8.1 Cider Compiler Errors

The Cider Compiler is responsible for taking a grammar specification, and producing Java source code tailored to the diagrammatic language that is specified. If the Cider format is used for specifying the grammar (this is the recommended format, which is used throughout this manual), the Cider Compiler will also have to first translate the specification into an XML format before processing of the grammar can take place.

The first set of problems that can occur are basic typographical errors; for example, characters may be missing or others may be present where they should not be, and names or keywords may be misspelt. In this situation, the Cider Compiler will report an error and indicate where in the input file the error occurred. The location of the error will typically be given as the line number in the XML file, even when the original specification was given in the Cider format. To help find the source of the problem in the original Cider format file, each XML file produced by the Cider Compiler includes comments indicating what line of the original file each XML component was based on.

Most other errors that the Cider Compiler may produce are related to the validity of the grammar itself. These errors can be simple to rectify (for example, if an illegal string has been used to define a symbol name), or fairly complex (for example, the grammar may not be stratified because of the presence of a negative dependency loop in the symbol dependency graph). In each case, the error produced by the Cider Compiler should give some basic detail on what has caused the problem (but will offer no advice as to how the problem might be rectified).

8.2 Java Compiler Errors

The Cider toolkits provides a very powerful grammar specification mechanism, but with this power also comes significant complexity. As a result, some parts of grammar specifications are not checked in depth by the Cider Compiler. In some cases, this is specifically because the Java compiler will effectively check them when the source code is compiled, and it would be redundant to duplicate this checking. In particular, this is the case with datatype checking; the Cider Compiler never does any checking with regard to the datatypes of symbol attributes.

Consider, for example, initializing and setting attribute values in productions (see Section 5.4.7). If an attribute of datatype `double` is assigned a `String` value in one of the production rules, the Cider Compiler will not detect this. However, the Java compiler will detect this, and report a compile time error. The same is true for the arguments of Java functions that are specified in the grammar (and for the functions themselves; they must be `static`, not throw any exceptions and the return datatype must match or Java compiler errors will be reported). Of course, the contents of any `@java` commands (defined in Section 5.4.2) are also left to the Java compiler to validate.

The main point to take away for application developers is that errors in grammar specifications may not be detected by the Cider Compiler, but may go on to produce a Java compiler error. Having said this, it is quite likely that many (if not most) Java compiler errors will be the result of application code that has nothing to do with the Cider components. Note also that care must be taken to ensure that the Java compiler is given a classpath that provides access to all the required classes.

8.3 Runtime Errors

It is possible that the Cider components of an SDE application behave unexpectedly or even cause the application to crash at runtime. The first point that should be noted is that any unexpected behaviour may not be due to the Cider components performing incorrectly. Even with moderately sized grammars, it is not always possible to see the full ramifications of how a specification will behave in certain situations. Unexpected behaviour may be due to subtleties in the grammar that were not foreseen when the specification was created. Even if this is the case, it would still be nice to have a way of finding out what the grammar is doing, so that errors in the specification can be corrected.

CIDER provides two main ways of monitoring the state of the parse forest as it changes during diagram creation and manipulation. The first is the `printForest()` method in the class `Interpreter`. This method provides a text-based "snap-shot" of the current structure of the parse forest. While useful however, often a static examination of the parse forest is not sufficient; sometimes it may be necessary to monitor in detail how the parse forest is dynamically changing as the result of user interactions.

To monitor dynamic changes, CIDER produces a runtime log file that details the changes being made to the parse forest as production rules are evaluated and rechecked and attribute values are kept up to date. The log is written to two rotating files (limited to 5 megabytes each) called `cider0.log` and `cider1.log`, which appear in the directory in which the application is run. Each log entry details not only what has happened at a given point in time, but also when it happened, and which of the CIDER components was responsible for it happening. Therefore, to explore why unexpected behaviour has occurred, the log files provide a useful resource. Note that amount of reporting that is written the log file can be altered by using the `setLoggingLevel()` method in the `Interpreter` class.

Of course, there is one further way in which the dynamic changes to the parse forest can be monitored, and that is programmatically in the application itself via the use of the `GrammarSymbolListener` interface (which like the `Interpreter` class is in the package `au.edu.monash.csse.tonyj.cider.interpreter`).

A final point to note is that there are certain aspects of a grammar specification for which CIDER expects the specification to "do the right thing." For example, the CIDER Compiler will issue a warning when a multi-production attribute graph loop dependency is detected (see Appendix B), but it will still proceed because the specification *might* be valid. However, if the specification in question is not valid, then runtime behaviour is undefined. Note that this could result in subtle errors that are difficult to detect, or possibly in more catastrophic behaviour such as an infinite loop or a runtime error that crashes the application.

9 The CIDER C API

This section describes the C API for CIDER which can be automatically generated by CIDER Compiler 0.3.1. A description of the API composition and structure is given, followed by an example C application.

The C API comprises a header file and corresponding implementation and is automatically generated by the CIDER Compiler. The generation of these files is invoked using the `-c` command-line switch of the CIDER Compiler. Given a CIDER grammar named `Example` in the file `Example.grammar` and an optional transformation file `Example.transformation`, two files containing C code are created: `ExampleInterpreter.c` and `ExampleInterpreter.h`. The latter file provides the C programmer's API, whilst `ExampleInterpreter.c` contains the implementation of the wrapper functions. A developer need not be concerned with the contents of `ExampleInterpreter.c` and should instead focus on the API defined in `ExampleInterpreter.h`. The CIDER C API supports abstract symbol definitions and inheritance, all integral Java datatypes, Strings, and Vectors of floating point values.

The `-b` command-line switch of the CIDER Compiler generates a file which is named `ExampleControl.c` given a grammar `Example`. This C code provides a skeleton for a controller which can be used to integrate CIDER with a C implementation. The skeleton controller contains callback functions which are dispatched when CIDER adds/removes/modifies a symbol, as well as functions to create a Java Virtual Machine, load the CIDER interpreter, etc. The `-b` switch should generally be used only once to create an initial starting point for the implementation of controller code. An existing `ExampleControl.c` file will not be over-written if the `-b` switch is given. This behaviour ensures that a working copy of the controller cannot be lost due to accidental use of the switch.

9.1 Macros

The beginning of the C header file contains a number of automatically generated macro definitions. Each macro is prefixed by the grammar name so that the API is tied clearly to a specific grammar. A

SYMBOL macro is defined for each symbol in the grammar specification. Each of these macros represents a particular type of terminal or non-terminal (but not abstract) symbol in the grammar. _SYMBOL_ macros are postfixed by the name of the CIDER symbol in uppercase. Macro definitions for each transformation are then listed and be identified by the _TRANSFORMATION_ substring. The transformation macros are derived from the optional transformation file; should this file not exist or exist in an empty state, no transformation macros will be shown. Each _TRANSFORMATION_ macro is postfixed by the label given to the transformation in the grammar specification. Finally, macros to represent CIDER logging levels are defined. These macros can be identified by the _LOGLEVEL_ substring and are intended for use with the _setLoggingLevel() function (discussed in Section 9.2).

The header file also defines two very important API types: an Interpreter type and a Symbol type. The names of these types are always prefixed with the grammar name (e.g. ExampleInterpreter and ExampleSymbol). Both of these types represent C structures. When initialized, an Interpreter structure contains all data relating to the CIDER interpreter. The Interpreter is used throughout the C API since all operations communicate with the run-time interpreter. The Symbol structure is used when adding, modifying or deleting CIDER symbols from the parse forest.

9.2 Standard Utility Functions

The macros defined in the API are used in conjunction with a standard set of utility functions. These functions are common to all APIs generated by CIDERĖach function is prefixed by the grammar name to avoid function redefinition between APIs. These functions are now discussed.

```
ExampleInterpreter * Example_create(JavaVM * jvm,
                                    void (*fatalError_callback)(const char * msg));
```

The _create() function creates an instance of the interpreter and returns a pointer to the aforementioned interpreter. A Java virtual machine (Java VM or JVM, see Section 9.4) must be passed as the first parameter, and a error handling function may also be specified as the second parameter. The remaining utility functions take a pointer to an interpreter as the first parameter.

```
void Example_destroy(ExampleInterpreter * interp);
```

The _destroy() function deallocates the interpreter which is given as the sole parameter. The destroy function serves to complement the _create() function above.

```
unsigned Example_getSymbolCount(
                 ExampleInterpreter * interp);
```

The _getSymbolCount function returns the number of symbols in the CIDER parse forest.

```
ExampleSymbol * Example_getSymbolAt(
                 ExampleInterpreter * interp,
                 unsigned pos);
```

The _getSymbolAt() function returns the symbol a position pos and is intended to be used in conjunction with _getSymbolCount(). A non-negative integer index must be passed as the second parameter.

```
jboolean Example_addTerminalSymbol(
                 ExampleInterpreter * interp,
                 ExampleSymbol * symbol);
```

The _addTerminalSymbol() function adds a terminal symbol to the CIDER parse forest and returns the success or failure of the operation. A terminal symbol must be passed as the second parameter.

```
jboolean Example_removeTerminalSymbol(
                 ExampleInterpreter * interp,
                 ExampleSymbol * symbol);
```

The `_removeTerminalSymbol()` function removes a terminal symbol from the CIDER parse forest and returns the success or failure of the operation. The CIDER terminal symbol to remove must be given as the second parameter. This function complements `_addTerminalSymbol()`.

```
jboolean Example_symbolExists(
                 ExampleInterpreter * interp,
                 ExampleSymbol * symbol);
```

The `_symbolExists()` function returns true if the CIDER symbol passed as the second parameter exists in the parse forest. The function returns false otherwise.

```
jboolean Example_applyTransformation(
                 ExampleInterpreter * interp,
                 int transformationId);
```

The `_applyTransformation` function applies the transformation rule indicated by the second parameter; the function returns true upon successful application and false otherwise. The particular transformation can be specified using one of the automatically generated `TRANSFORMATION_` macros found in the top portion of the header file. Transformations applied using this function will not occur in a parallel fashion.

```
jboolean Example_applyTransformationInParallel(
                 ExampleInterpreter * interp,
                 int transformationId);
```

The `_applyTransformationInParallel()` function performs the same role as `_applyTransformation()` with the exception of applying transformations in parallel.

```
int Example_getSymbolType(
                 ExampleInterpreter * interp,
                 ExampleSymbol * symbol);
```

The `_getSymbolType()` function returns an integer which represents the type of the CIDER symbol passed as a second parameter. The integer returned should be compared to the automatically generated `_SYMBOL` macros found in the top portion of the header file.

```
int Example_getSymbolId(
                 ExampleInterpreter * interp,
                 ExampleSymbol * symbol);
```

The `_getSymbolId()` function returns an integer which identifies uniquely the CIDER symbol passed as the second parameter.

```
jboolean Example_isSymbolTerminal(
                 ExampleInterpreter * interp,
                 ExampleSymbol * symbol);
```

The `_isSymbolTerminal()` function returns true if the CIDER symbol passed as the second parameter is a terminal symbol. The function returns false otherwise.

```
const char * Example_getSymbolName(
                 ExampleInterpreter * interp,
                 ExampleSymbol * symbol);
```

The `_getSymbolName()` function returns the textual name given to the CIDER symbol passed as the second parameter. The name is represented as a C character string.

```
const char * Example_symbolToString(
                 ExampleInterpreter * interp,
                 ExampleSymbol * symbol);
```

The `_symbolToString()` function returns a C string representation of the CIDER symbol passed as the second parameter. The string returned contains the name of the symbol, and pair of attribute labels with corresponding attribute values.

```
const char * Example_forestToString(
                 ExampleInterpreter * interp);
```

The `_forestToString()` function returns the entire parse forest represented as a C string. This function is useful for debugging applications which use the C CIDER API.

```
void Example_setAutoForestChecking(
                 ExampleInterpreter * interp,
                 jboolean isOn);
```

The `_setAutoForestChecking()` function is used to turn CIDER automatic forest checking on and off. A `true` value for the second parameter will switch forest checking on; a `false` parameter value will turn forest checking off. It is useful to turn forest checking off prior to modifying more than one constrained attributes of a CIDER symbol. Generally in these cases, forest checking is turned off, the attribute values are modified, and forest checking is re-enabled prior to the last modification. In this way, the constraint solver will not attempt to optimize the modification of each parameter separately. Rather, the changes are handled together.

```
jboolean Example_isAutoForestCheckingOn(
                 ExampleInterpreter * interp);
```

The `isAutoForestCheckingOn()` function returns true if the CIDER automatic forest checking is turned on, and returns false otherwise. This function is intended for use in conjunction with `_setAutoForestChecking()`.

```
void Example_setLoggingLevel(
                 ExampleInterpreter * interp,
                 int logLevel);
```

The `_setLoggingLevel()` function uses its second parameter to sets the logging granularity of the CIDER interpreter. The second parameter to this function should be set using the automatically generated `_LOGLEVEL_` macros found near the beginning of the header file.

```
JNIEnv * Example_getEnv(
                 ExampleInterpreter * interp);
```

The `_getEnv()` function returns the JNI environment structure given the interpreter as a sole parameter. This is a utility function and is not expected to be used often.

```
void Example_set_symbolAdded_callback(
                 ExampleInterpreter * interp,
                 ExampleCallback callback);
```

`_set_symbolAdded_callback()` registers a callback function which is called in the event of CIDER adding a symbol to the parse forest. The callback function must be given as the second parameter.

```
void Example_set_symbolRemoved_callback(
                 ExampleInterpreter * interp,
                 ExampleCallback callback);
```

`_set_symbolRemoved_callback()` registers a callback function which is called in the event of CIDER removing a symbol from the parse forest. The callback function must be given as the second parameter.

```
void Example_set_symbolModified_callback(
                 ExampleInterpreter * interp,
                 ExampleCallback callback);
```

`_set_symbolModified_callback()` registers a callback function which is called in the event of CIDER modifying an existing symbol in the parse forest. The callback function must be given as the second parameter.

9.3 Functions for Symbol Manipulation

The remainder of the C API is generated according the symbol definitions contained in the CIDER grammar file. For each terminal and non-terminal (but not abstract) symbol, a `_create()` function is defined by the API. The name of this function is formed by concatenating the grammar name, symbol name, and `_create()` strings. The `_create()` function returns a pointer to a new symbol, the type of which is indicated by the full function name. Parameters to initialize the value of each symbol attribute are specified in the order indicated by the grammar file. All built-in JNI types are facilitated and two special cases are also implemented: strings and vectors of Floats. Table 1 lists each Java attribute type and its corresponding type to be used with the C API.

Java attribute type	C API type
`boolean`	`jboolean`
`int`	`jint`
`float`	`jfloat`
`double`	`jdouble`
`char`	`jchar`
`java.lang.String`	`char *`
`java.util.Vector`	`float *`

Table 1: Java attribute types and the C API equivalents

String attributes are set using a pointer to a C character array. Java `Vector` attributes are emulated by one-dimensional arrays in the C API. Consequently, `Vector` attributes must be set using two parameters: a pointer to an existing array of floats, and an non-negative integer to indicate the size of the C array. For each `Vector` attribute, the integer parameter directly follows the corresponding array pointer.

Consider a CIDER symbol with the definition:

```
Gesture (terminal)
    visible:boolean, points:java.util.Vector,
    name:java.lang.String, x:int, y:int;
```

The following example shows the symbol `_create()` prototype for this symbol:

```
ExampleSymbol * Example_Gesture_create(
                ExampleInterpreter * interp,
                jboolean visible,
                float * points, int points_size,
                char * name,
                jint x, jint y);
```

As with all API functions, a pointer to the CIDER interpreter must be passed as the first parameter. Next, a `jboolean` is used to indicate whether the gesture is visible on the screen. The sequence of gesture point coordinates is then passed using a pointer to an array and an integer indicating the array size. The name of the gesture is next given in the form of a C string and, finally, two JNI integers are used to specify the initial on-screen position of the gesture. Memory for the character and float arrays must be created by the C application programmer.

A pair of `_get` and `_set` functions are also created for each symbol attribute. Each function name is formed using the grammar name as a prefix, followed by the substring `_get_` or `_set_` and the label of the attribute being modified or retrieved. Both `_get` and `_set` methods require a pointer to an interpreter as the first parameter. Generally, the `_get` methods take no additional parameters and return the attribute value in a C equivalent type as per Table 1. The exception to this case is Vector typed attributes. Here, a pointer to an integer variable must be passed as the second function parameter so that the size of the returned float array can be obtained. As before, it is the application programmer's responsibility to free the memory used by returned character and float arrays. Note that the return type of constrained attributes is always `jdouble`.

Attribute `_set` methods generally take a single second parameter to specify the value that the attribute should take. These attributes are of the C equivalent types for the Java attributes as shown in Table 1. As before, `Vector` typed attributes are a special case; functions to set these attributes take a pointer to an array of floats as the second parameter. An additional integer parameter is required to indicate the size of the array.

9.4 A Worked Example

Assume the following CIDER grammar named `Example` which features three symbols and one production.

```
Line (terminal)
    name:java.lang.String,
    x_start:@cv[10, 1000], y_start:@cv[10, 1000],
    x_end:@cv[10, 1000], y_end:@cv[10, 1000];

Text (terminal)
    label:java.lang.String;

Arrowhead (terminal)
    points:java.util.Vector, angle:@cv[10, 1000],
    x_pos:@cv[10, 1000], y_pos:@cv[10, 1000];

Arrow (non-terminal)
    x_start:@cv[10, 1000], y_start:@cv[10, 1000],
    x_end:@cv[10, 1000], y_end:@cv[10, 1000];


:production {
    a:Arrow ::= l:Line, t:Arrowhead
    where {
        @sc(4.0, "-[l.x_end] + [t.x_pos] == 0.0");
        @sc(4.0, "-[l.y_end] + [t.y_pos] == 0.0");
    }
    set {
        a.x_start := @ce("[l.x_start]");
        a.y_start := @ce("[l.y_start]");
        a.x_end   := @ce("[l.x_end]");
        a.y_end   := @ce("[l.y_end]");
    }
}
```

Also assume the following transformation to produce a new Text symbol from any Arrow which ends at coordinates $(250, 500)$.

```
arrowToLine:transformation {
    i:Arrow
    where {
        i.x_start == 250;
        i.y_start == 500;
    }
    apply {
        (nts) text:Line("A line was transformed.");
    }
}
```

An example C program is now developed using the API generated from the grammar and transformation specifications. First, include headers for C input/output and other common routines, along with the automatically generated, grammar-specific header file.

```
#include <stdio.h>
#include <stdlib.h>
#include "ExampleInterpreter.h"
```

Next, define callback functions which are called when CIDER adds, removes, or modifies a symbol to the parse forest. These functions simply output the memory address of the CIDER symbol and the symbol name. In the case of an added symbol, a `1` is output if the symbol is terminal; `0` is output if the symbol is a non-terminal.

```
void symbolAdded(ExampleInterpreter * interp,
                 ExampleSymbol * symbol)
{
    printf("Symbol added: %x of type %s\n",
           symbol,
           Example_getSymbolName(interp, symbol));

    printf("Symbol is terminal: %d\n",
           Example_isSymbolTerminal(interp, symbol));
}

void symbolRemoved(ExampleInterpreter * interp,
                   ExampleSymbol * symbol)
{
    printf("Symbol removed: %x of type %s\n",
           symbol,
           Example_getSymbolName(interp, symbol));
}

void symbolModified(ExampleInterpreter * interp,
                    ExampleSymbol * symbol)
{
    printf("Symbol modified: %x of type %s\n",
           symbol,
           Example_getSymbolName(interp, symbol));
}
```

The following function creates a Java Virtual Machine (JVM) and returns a pointer to the JVM. This function is almost entirely generic and reusable; the only grammar-specific specific code concerns the Java class path which must include the Java interpreter `.jar` file. In this case, the CIDER compiler has generated an interpreter and packaged it in `Example.jar`.

```
JavaVM * createJVM(void)
{
    JavaVM * jvm = 0;
    JNIEnv * env = 0;
    JavaVMInitArgs vm_args;
    JavaVMOption options[1];

    options[0].optionString =
    "-Djava.class.path=.:CiderCore-0.3.1.jar:Example.jar";

    vm_args.version = JNI_VERSION_1_4;
    vm_args.options = options;
    vm_args.nOptions = 1;
    vm_args.ignoreUnrecognized = JNI_TRUE;

    jint res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);

    if (res < 0)
    {
        printf("Fatal Error: Couldn't create a Java VM\n");
        exit(1);
    }

    return jvm;
}
```

Next, define a function to deallocate a JVM. This function takes a pointer to a JVM and is generic.

```
void destroyJVM(JavaVM * jvm)
{
    jint res = (*jvm)->DestroyJavaVM(jvm);

    if (res < 0)
    {
        printf("Fatal Error: Couldn't destroy the Java VM\n");
        exit(1);
    }
}
```

The main() function is now defined. The function begins by creating a JVM and declaring a pointer return_array to an array which will be obtained later. A static array of five floats is also declared along with an integer to store array sizes. The variables will be used to set the value of an Arrowhead's Vector attribute.

```
int main(void)
{
    int i;
    JavaVM *jvm = createJVM();

    float *return_array,
           test_array[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
    int array_size;
```

The next step involves creating an instance of the CIDER interpreter using the JVM created above. The default error function is used in this example, as indicated by the NULL parameter to Example_create(). The CIDER logging level is then set to a verbose mode and, following this, the three callback functions defined earlier are registered using the loaded interperter.

```
    ExampleInterpreter * interp = Example_create(jvm, NULL);
    Example_setLoggingLevel(interp, Example_LOGLEVEL_SEVERE);

    Example_set_symbolAdded_callback(interp, symbolAdded);
    Example_set_symbolRemoved_callback(interp, symbolRemoved);
    Example_set_symbolModified_callback(interp, symbolModified);
```

Having loaded the interpreter and registered callback functions, some terminal symbols can now be created. The following segment of code creates Line and Arrowhead symbols and adds them to the CIDER parse forest. Note the use of a string literal to set the line label. Also note the use of a C array to set the initial value for the arrowhead points attribute. The size of the array is specified by an integer parameter which proceeds the array parameter. The last line of this segment outputs the CIDER parse forest, which should now contain the Line and Arrowhead symbols.

```
    ExampleSymbol * line = Example_Line_create(
                           interp, "Example line",
                           100, 100, 90, 100);

    ExampleSymbol * arrowhead = Example_Arrowhead_create(
                           interp, test_array,
                           5, 45, 50, 80);

    Example_addTerminalSymbol(interp, arrowhead);
    Example_addTerminalSymbol(interp, line);

    printf("Forest:\n%s\n", Example_forestToString(interp));
```

The next segment of code shows how the value of a Vector attribute can be returned as a C array. The code first gets the value of the points attribute and outputs the contents of this array. Notice that the array memory is explicitly reclaimed after this operation. The values of the static array created earlier are then changed and the _set function is called to reflect these changes in the CIDER symbol. The Vector values are extracted and printed to show that the change has taken affect.

```
    return_array = Example_Freehand_get_points(interp,
                                              freehand,
                                              &array_size);
```

```
for(i = 0; i < array_size; i++)
    printf("Return array[%d]: %f\n", i, return_array[i]);

free(return_array);

test_array[0] = 6.0; test_array[1] = 7.0;
test_array[2] = 8.0; test_array[3] = 9.0;
test_array[4] = 10.0;
Example_Freehand_set_points(interp, freehand,
                            test_array, 5);

return_array = Example_Freehand_get_points(interp,
                                           freehand,
                                           &array_size);

for(i = 0; i < array_size; i++)
    printf("Return array[%d]: %f\n", i, return_array[i]);

free(return_array);
```

The next block of C code switches automatic forest checking off before modifying the end-point coordinates of the Line symbol. Forest checking is reactivated prior to the last `_set` operation. Forest checking should be switched on and off in this manner whenever more several attributes of a symbol must be modified. This ensures that CIDER evaluates productions only after the series of attribute modifications is complete. The new coordinates of the Line's end-point mean that the production specified earlier applies and, consequently, CIDER will create a new non-terminal Arrow symbol.

```
Example_setAutoForestChecking(interp, false);
Example_Line_set_x_end(interp, line, 50);
Example_setAutoForestChecking(interp, true);
Example_Line_set_y_end(interp, line, 80);
```

We now modify the start coordinates of the Line such that the transformation becomes applicable. Again, automatic forest checking is turned off prior to setting the first attribute value, and switched back on before setting the last `_set` operation. The remaining code in the block is responsible for outputting details about each symbol in the parse forest. The last line of C code in this block should indicate that the line symbol still exists at this point in time.

```
Example_setAutoForestChecking(interp, false);
Example_Line_set_x_start(interp, line, 200);
Example_setAutoForestChecking(interp, true);
Example_Line_set_y_start(interp, line, 200);

printf("\nSymbols BEFORE transformation:\n");

for(i = Example_getSymbolCount(interp) - 1; i >= 0; --i)
{
    ExampleSymbol * s = Example_getSymbolAt(interp, i);
```

```
        printf("sym: %x of type %s, id %d, is terminal %d\n",
               s,
               Example_getSymbolName(interp, s),
               Example_getSymbolId(interp, s),
               Example_isSymbolTerminal(interp, s));
    }

    printf("Line exists: %d\n", Example_symbolExists(interp, line));
```

The first line of the following code segment applies the transformation rule defined earlier. Since the transformation can be applied given the current state of the parse forest, a new Text symbol will be created and the Arrow symbol (along with its constituent Line and Arrowhead symbols) will be deleted. As before, the remaining code outputs information about the CIDER parse forest. This output will reflect the changes due to the transformation - only the Text symbol produced by the transformation should be present in the parse forest. Thus, the final line of C in this block will output a `0` to indicate that the initially created Line symbol no longer exists.

```
    Example_applyTransformation(interp, Example_TRANSFORMATION_arrowToLine);

    for(i = Example_getSymbolCount(interp) - 1; i >= 0; --i)
    {
        ExampleSymbol * s = Example_getSymbolAt(interp, i);

        printf("sym: %x of type %s, id %d, is terminal %d\n",
               s,
               Example_getSymbolName(interp, s),
               Example_getSymbolId(interp, s),
               Example_isSymbolTerminal(interp, s));
    }

    printf("Line exists: %d\n", Example_symbolExists(interp, line));
```

Finally, we deallocate the CIDER interpreter and JVM in turn, and the program terminates.

```
    Example_destroy(interp);
    destroyJVM(jvm);

    return 0;
}
```

A Stratified Grammars

In order to understand the role of stratified grammars with regard to CIDER, let us first briefly review the concept of symbol dependencies.

When a grammar is specified, the output symbols of each production will be dependent on other symbols in the grammar for their formation. The most straight-forward type of dependency is a reduction dependency; for every production, each output symbol depends on each input symbol being present in order for it to be generated. Since the input symbols *must* be present for the output symbols to be generated, they are examples of *positive* symbol dependencies.

There can also be other symbols that must be present, as defined by positive existential conditions (specified in the grammar using the `exists` keyword inside a `production` definition; see Section 5.4.5). As these symbols also must be present, they are also examples of positive symbol dependencies.

However, there can also be *negative* symbol dependencies. These arise through the specification of negative existential conditions (specified in the grammar using the `not_exists` keyword inside a `production` definition; again, see Section 5.4.5). Negative existential conditions require that certain symbols of a specific type must *not* be present in the parse forest in order for the output symbols of a production to be generated. It is these negative symbol dependencies that will be the focus of this section. (Note that if a `not_exists` command is nested within another `not_exists` command, the two negations will cancel each other out and the inner existential symbol will represent a positive symbol dependency.)

There is one other way in which a negative symbol dependency can occur in a grammar. When a production is defined for which the input symbol list contains a *symbol collection* (see Section 5.4), there is an implicit negative symbol dependency for that symbol type (as well as a positive symbol dependency). This is because the symbol collection is invalidated whenever a symbol of the collection type is created or destroyed.

CIDER will only work with stratified constraint multiset grammars. A stratified CMG is one that contains no negative symbol dependency loops [10]. To illustrate what a negative symbol dependency loop is, consider the following simple production (Example 1) where `A` is a `non-terminal` symbol with no attributes, and `B` is a `terminal` symbol with no attributes.

```
:production {
   a:A ::= b:B
   where {
      not_exists a2:A
      where true;
   }
}
```

If a new symbol of type `B` is added to an empty parse forest, then this production will be evaluated. Since the forest contains no instances of symbol `A`, the conditions of the production will be met and the production will be successfully applied. However, successful application of the production will result in an instance of symbol `A` being added to the parse forest. This will then immediately invalidate the production, which in turn will see the instance of symbol `A` removed from the parse forest. However, with no more instances of symbol `A` present, the conditions of the production can be met again, and thus it can be reapplied.

It should be clear that this production will lead to an infinite loop, with the production being continuously applied and then immediately invalidated. Thus, the above production is a simple example of a negative symbol dependency loop.

While the above example shows a negative symbol dependency loop contained within a single production, it is also possible for such loops to occur across multiple productions. For example, consider the following three productions (Example 2) involving two `terminal` symbols, `M` and `N`, and three `non-terminal` symbols, `P`, `Q` and `R`. Again, none of the symbols have any attributes.

```
:production {
   p:P ::= m:M
   where {
      not_exists r:R
      where true;
   }
}
```

```
:production {
   q:Q ::= n:N
   where {
      exists p:P
      where true;
   }
}
```

```
:production {
   r:R ::= q:Q
   where true;
}
```

If the parse forest contains only an instance of `N`, and then an instance of `M` is added, an infinite loop will occur with the continuous creation and then destruction of instances of the symbols `P`, `Q` and `R`.

The negative symbol dependency loops are easier to see to when a symbol dependency graph is created. Figure 9 illustrates the symbol dependency graphs for both the one production (Example 1) and three production (Example 2) scenarios discussed in this section. The blue arrows represent positive symbol dependencies (either from reductions or positive existential conditions), while the red arrows represent negative symbol dependencies (from negative existential conditions).

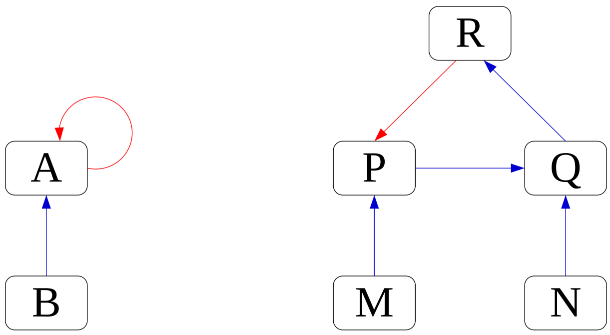


Figure 9: The symbol dependency graphs for both the Example 1 and Example 2 production systems.

Note that the negative symbol dependency loops must form a *directed* path in the symbol dependency graph, where only one of the components of that path need be a negative dependency (as is the case in the Example 2 symbol dependency graph where two of the three path components in the loop represent positive dependencies). Also, note that if the examples had contained symbols with attributes, this would have no effect on the symbol dependency graphs.

To check for all possible negative symbol dependency loops, the CIDER Compiler constructs a symbol dependency graph as it processes a grammar specification. Any negative symbol dependency loops that are present in a grammar will be detected by the CIDER Compiler and reported to the user as an error.

B CMG Attributes in CIDER

Table 2 illustrates how attribute values are set in the CMGs used by CIDER (note that the notation used in the examples is for illustrative purposes only – it is not the same as the notation that CIDER uses). The scenarios consider the situation where two terminal symbols, `A` and `B` (with attributes `x` and `y` respectively), have produced a non-terminal symbol `C` (with an attribute `z`). The values of the terminal symbol attributes will have been initialized before the production of the non-terminal. The first 10 scenarios examine different ways in which the attributes values can be set, showing simple examples of both valid settings as well as illegal settings. The final (11th) scenario shows an illegal attribute setting for a parse tree that has an extra non-terminal symbol `D` (with an attribute `w`), that is produced from symbol `C`.

Shaded boxes in the attribute dependency graphs indicate attributes whose value depends on the

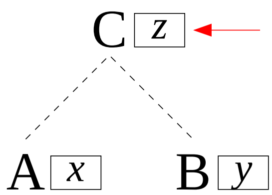
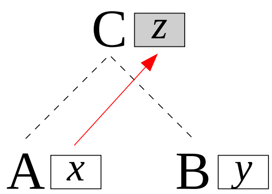
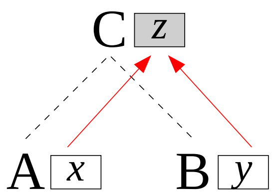
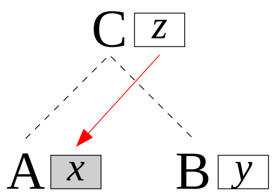
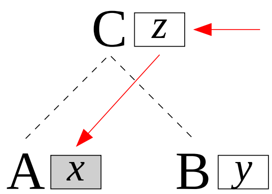
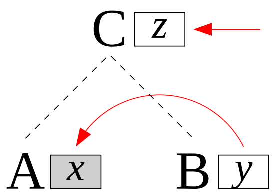
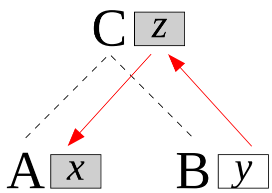
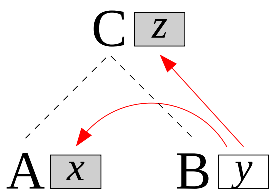
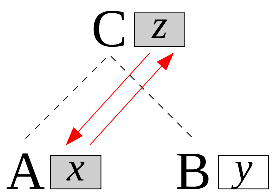
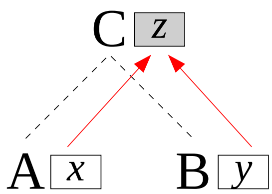
Description	**Attribute Settings**	**Attribute Dependency Graph**
Scenario 1 Initialize attribute with a constant value (No dependencies)	`C.z := 3` *(initialize)*	
Scenario 2 Copy the value of another attribute	`C.z := A.x`	
Scenario 3 Set the attribute value using a function that depends on other attribute values	`C.z := A.x + B.y`	
Scenario 4 – ***Illegal*** Cannot use an uninitialized output symbol attribute to set an input symbol attribute	`A.x := C.z`	
Scenario 5 Initialize output symbol attribute and then use it to set input symbol attribute	`C.z := 4` *(initialize)* `A.x := C.z`	

Table 2: Various scenarios for setting attribute values.

Description	Attribute Settings	Attribute Dependency Graph
Scenario 6 Use input symbol attribute value to set another input symbol attribute	`C.z := 7` *(initialize)* `A.x := B.y`	
Scenario 7 – ***Illegal*** Cannot have attribute used as both a setter and a settee in the same production	`A.x := C.z` `C.z := B.y`	
Scenario 8 There is always a valid alternative to Scenario 7 that has the same effect	`A.x := B.y` `C.z := B.y`	
Scenario 9 – ***Illegal*** Cannot have loops formed by attribute setting within a production	`A.x := C.z` `C.z := A.x`	
Scenario 10 – ***Illegal*** Cannot have a specific attribute set more than once	`C.z := A.x` `C.z := B.y`	

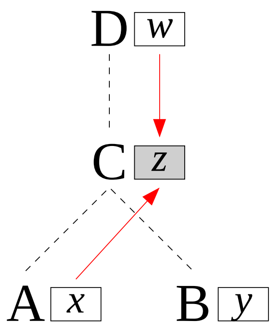
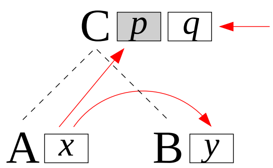
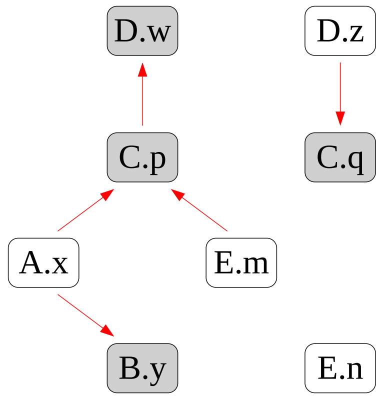
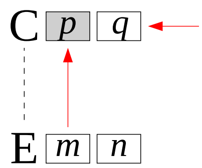
Scenario 11 – ***Illegal***
A single attribute cannot be set by two productions simultaneously
	Upper Production
	`C.z := D.w`
	Lower Production
	`C.z := A.x`
		

Table 2: Various scenarios for setting attribute values.

Attribute Settings	**Production Attribute Dependency Graph**	**Grammar Attribute Dependency Graph**

Production 1
`C.q := 4` *(initialize)*
`C.p := A.x`
`B.y := A.x`
		

Production 2 `C.q := 3` *(initialize)* `C.p := E.m`		

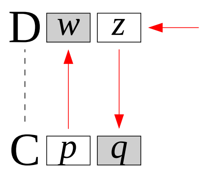
Production 3
`D.z := 8` *(initialize)*
`D.w := C.p`
`C.q := D.z`
		

Table 3: Attribute dependency graph for an example three production grammar.

values of other attributes. The red arrows indicate the directions of these dependencies. Considered from a dynamic perspective, this means that attributes in unshaded boxes can have their values changed at a later time, whereas attributes in shaded boxes cannot. When an unshaded attribute is modified, any other attributes that depend on it (as indicated by the red arrows) also have their values updated automatically.

Cider does not do any attribute dependency graph checking at runtime (to avoid the extra computational expense). Rather, attribute dependency graph checking is done by the Cider Compiler. This means that the attribute dependency graph constructed is a generalization based on the entire grammar, rather than specific to a particular parse tree.

The Cider Compiler will report an error if any of the illegal scenarios listed in Table 2 are detected in the input specification. The Cider Compiler will also check for loop dependencies that exist between attribute settings specified across multiple productions (Scenario 9 illustrates a loop within a single production). However, there is a limitation resulting from the attribute dependency graph being based on a generalization of the entire grammar and not a specific parse tree – some loop dependencies detected may not actually be illegal (in such cases, the grammar will never lead to a parse tree that actually contains an attribute setting loop). Thus, the Cider Compiler will only issue a warning when a multi-production attribute loop dependency is detected – it is up the person specifying the grammar to ensure that no attribute setting loops will actually occur when the grammar is used.

Table 3 gives an example of a grammar attribute dependency graph derived from a simple three production rule grammar (with terminal symbols `A`, `B` and `E`, non-terminals `C` and `D`, and attributes as indicated in the table). The grammar attribute dependency graph shows the dependency relationships between all of the attributes of all of the symbols. Shaded ovals indicates attributes that are dependent on the value of other attributes (as indicated by the red arrows). Just as grammars have terminal and non-terminal symbols, we can also consider attributed grammars to have terminal and non-terminal attributes (indicated by unshaded and shaded ovals respectively in this example).

Note that in the Cider toolkit, it is possible for the user to modify the values of terminal attributes at any time (using a `set_`*attribute_name*`()` method). However, it is not possible for the user to modify the values of non-terminal attributes – only the Cider Interpreter is permitted to do that (the `set_`*attribute_name*`()` method for such attributes is not publicly available). This is true even for non-terminal attributes that belong to terminal symbols (such as `B.y` in this example); such attributes can be given an initial value when the terminal symbol is created, but cannot be modified by the user after that.

The only exception to this rule is for attributes that are constraint variables. These variables are controlled by a multi-way constraint solver and thus the `set_`*attribute_name*`()` method is always publicly available for them. It is possible to allow this without possible parse forest inconsistencies because the `set_`*attribute_name*`()` method for constraint variables only *suggests* a new value for the attribute – it is up to the multi-way constraint solver to determine if the value suggested is to be assigned to the attribute. Hence, it should be noted that in some cases, the `set_`*attribute_name*`()` method for a constraint variable may not change the value of that variable. (For example, this is likely to be the case if there is a production rule constraint involving both a constraint variable and another attribute that is not a constraint variable.) Thus, one should always use the `get_`*attribute_name*`()` method to obtain the value of a constraint variable, and never assume that the value passed in the most recent call to the `set_`*attribute_name*`()` method was assigned.

One further point of interest with regard to attribute setting is what happens when a production rule becomes invalidated. This is only of concern in situations such as Scenario 5 in Table 2, where the value of an output symbol has been used to set an input symbol attribute. When such a production rule is first evaluated, the input symbol attribute has its original value replaced with the value specified in the production's attribute settings. The original value is stored, however, so that if the production is later invalidated, the original value can be restored. Thus, it is as if the production never occurred. This is illustrated in Figure 10.

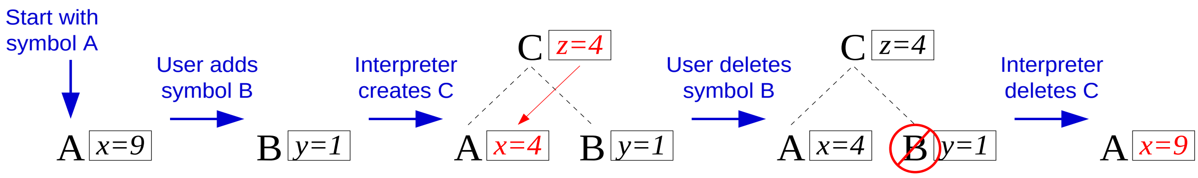


Figure 10: Example of restoration of an attribute value.

C Drawable Non-Terminal Symbols

The CIDER toolkit allows both terminal and non-terminal symbols to have a graphical representation. A graphical representation for a symbol is specified using the *Drawable Class* attribute within the corresponding symbol definition in a grammar specification (see Section 5.3).

While it is expected that most terminal symbol definitions will have a Drawable Class attribute, this is not common for a non-terminal symbol definition. However, it is permitted. There are just a few things that one must keep in mind when allowing a non-terminal symbol to be drawable.

The main point to be aware of is that every drawable symbol in the parse forest will have its `draw()` method called, regardless of where it is in the parse forest structure (the `draw()` method is defined in the interface `DrawableToken`, in the package `au.edu.monash.csse.tonyj.cider.canvas`). So if a drawable non-terminal of type `C` is produced from two drawable terminals of type `A` and `B`, all three of `A`, `B` and `C` will be drawn. Note that the `InterpretedTokenCanvas` is not aware of any structural relationship between any of the graphical symbols. Only the `Interpreter` is aware of this. The `InterpretedTokenCanvas` will simply draw the symbols, in whatever order they appear in the canvas stack (this order can be manipulated by the application developer; see the Javadocs for the `TokenCanvas` class for more details).

In some cases, it might be desirable for the graphical representation of a non-terminal symbol to *replace* those of the terminal symbols that compose it. The simplest way to achieve this is via the use of attributes. For example, the terminal and non-terminal symbols in question can each be given a `boolean` attribute called `isVisible`. Whenever a terminal symbol is created its `isVisible` attribute is initially set to `true`. Then, whenever the production that produces the non-terminal symbol is successfully evaluated, it will set the `isVisible` attribute of the terminal symbols to `false`, while also setting the `isVisible` attribute of the non-terminal symbol to `true`. This is illustrated by the following production.

```
:production {
   c:C ::= a:A, b:B
   where {
      ...
   }
   set {
      c.isVisible := true;
      a.isVisible := false;
      b.isVisible := false;
      ...
   }
}
```

All that is required then is that the `draw()` method associated with each symbol checks the value of the `isVisible` attribute, and only proceeds to draw the symbol if the attribute value is `true`. Of course, if the production that produced the non-terminal symbol is invalidated later, the `isVisible` attribute of the terminal symbols will automatically return to `true`, and thus they will be redrawn. Figure 11 shows the parse forest both before and after the above production has been applied.

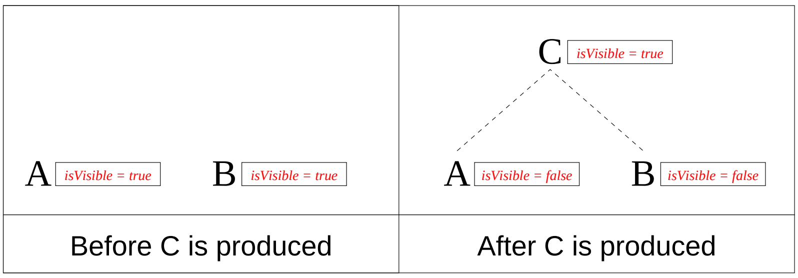


Figure 11: Parse forest state that results from applying the above example production.

D Interaction between the multi-way and one-way solvers

One interesting issue arising with the use of a multi-way constraint solver (Qoca) is how it cooperates with Cider's built-in one-way constraint solver when attribute values are updated. Ideally, every constraint that is involved in structure-preservation should be handled by the multi-way solver. If this is not the case, Cider adds additional multi-way constraints for equations handled by the one-way solver to maintain consistency (ensuring that none of the dependent one-way variables can be changed by the multi-way solver). For instance, consider the parse tree shown in Figure 12.

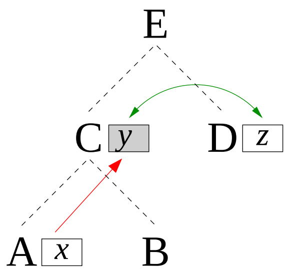


Figure 12: A simple parse tree example involving three terminal symbols, `A`, `B` and `D`, and two non-terminals, `C` and `E` (where `A`, `C` and `D` have attributes `x`, `y` and `z` respectively).

In this example, the symbols `C` and `E` are non-terminals, while `A`, `B` and `D` are terminals (where `B` and `E` have no attributes and the other symbols have attributes as shown). The lower production that generates symbol `C` includes the following attribute setting (illustrated by the red arrow in Figure 12).

```
C.y := A.x;
```

This attribute setting is passed to the one-way constraint solver. The upper production that generates symbol `E` contains the following constraint (indicated by the double-headed green arrow in Figure 12).

```
C.y == D.z
```

This constraint is controlled by the multi-way constraint solver (Qoca).

When symbol `E` was created, the values of `A.x`, `C.y` and `D.z` must have all been the same to ensure that the parse forest was in a consistent state (let us suppose they were all equal to 3.0). However, if the value of the attribute `D.z` is now modified to equal 5.0, the constraint solver will also modify the value of `C.y` (to also equal 5.0) so that the constraint in the upper production remains satisfied. This, however, would leave the parse tree in an inconsistent state, as the values of `A.x` and `C.y` would no longer match. (Note that the one-way attribute setting in the lower production will allow `C.y` to be automatically updated if `A.x` is modified, but not vice versa.)

The solution is to allow the attribute setting in the lower production to be controlled by the multi-way constraint solver (via the `@ce` command; see Section 5.4.7), making it equivalent to the following.

```
C.y == A.x
```

This would mean that when `D.z` is modified, the constraint solver will automatically modify both `C.y` and `A.x` to the same value in order to satisfy all of its constraints. This then guarantees that the parse forest will remain consistent.

Note that in the above example, if the constraint in the upper production is in the multi-way constraint solver, but the attribute setting in the lower production is not, then Cider has its own less flexible method for preventing inconsistencies in the parse forest. Each time the value of `A.x` is set (say to 2.0), Cider will not only modify `C.y`, but it will also add a constant constraint to the multi-way solver which fixes `C.y` at that value (in this case, `C.y == 2.0`). Then, if an attempt is made to modify `D.z`, this will be rejected by the multi-way solver as the new value will not allow all of its constraints to be satisfied. The only time that the constant constraint can be overridden is if `A.x` is changed to another value, in which case Cider will remove the old constant constraint, and once `C.y` is updated again (which should automatically result in `D.z` also being updated), a constant constraint for the new value of `C.y` will be added to the solver.

E Command Line Options for the CIDER Compiler

The CIDER Compiler takes a grammar specification as its input, and produces Java source code that is tailored to the diagrammatic language that has been specified. This code is then used in conjunction with the CIDER Core Components, allowing an application to be created that can process and interpret the specified diagrammatic language more efficiently.

The CIDER Compiler is run from the command line as follows.

```
java  -jar  CiderCompiler-0.3.1.jar
```

The CIDER Compiler has several command line options. The simplest of these is the help option.

`-h`	Help. This option lists the available command line arguments.

Note that when the help option has been included in the command line, all other options will be ignored. The help information will also appear if the CIDER Compiler is run with no command line arguments.

In normal operation, the purpose of the CIDER Compiler is to produce Java source code based on an input specification. The input specification comes in two components: a grammar file which *must* be present, and an optional transformations file. The CIDER Compiler receives the name of the required grammar file using one (and only one) of the following two arguments.

`-g` *grammar_file*	Specify the name of the grammar file (in CIDER format).
`-gx` *grammar_file*	Specify the name of the grammar file (in XML format).

If, optionally, transformation rules have also been specified, the CIDER Compiler will need to receive the name of the transformations file using one (and only one) of the following two arguments.

`-t` *transformations_file*	Specify the name of the transformations file (in CIDER format).
`-tx` *transformations_file*	Specify the name of the transformations file (in XML format).

The CIDER Compiler does all of its processing of input specifications using XML. However, since XML results in far more cumbersome specifications, the CIDER format (which is the one described in this document) was developed. It is recommended that the CIDER format be used. When receiving a specification in CIDER format, the CIDER Compiler will automatically translate it into the XML format before processing it. In doing so, it will also generate a new file with the same name as the original file, plus a `.xml` extension. So if a grammar is specified in CIDER format in a file called `mygrammar.txt`, the CIDER Compiler will automatically create a file called `mygrammar.txt.xml` containing an XML version of the grammar (if this file already exists, it will be overwritten). Obviously, if the grammar is supplied in the XML format, no translation will be required.

Note that if the specification includes both a grammar file and a transformations file, the two files can be in different formats; that is, the grammar file can be input in the CIDER format (using the `-g` option) while the transformations file is input as XML (using the `-tx` option), or vice versa.

The CIDER Compiler can also be instructed to generate a C API so that CIDER can be integrated with C and C++ programs. This API is implemented using the Java Native Interface (JNI). The compiler can also generate automatically a skeleton program which contains code to assist a developer aiming to integrate CIDER with a C/C++ application. Note that the skeleton code can only be generated when the C API is also requested.

`-c`	Generate a C API for the specified grammar.
`-b`	Generate a skeleton file to accompany the C API.

There are two final command line options.

`-translate_only`	Translate CIDER format files into XML without processing further.
`-w`	Suppress CIDER Compiler warnings.

The first of these options, `-translate_only`, can be used if the grammar file and/or transformations file are in the Cider format. If this is the case, the Cider format files are translated into XML (creating `.xml` versions of the files) without the Cider Compiler processing the specification further (meaning no Java source code will be generated). The second option, `-w`, will prevent the Cider Compiler from issuing any warning messages during compilation of the input specification.

For the Cider Compiler to successfully process input specifications, it must have access to the Cider DTD file (called `Cider-0.3.1.dtd`), which must be placed in the same directory as the grammar file and transformations file being used. In order for the Cider Compiler to generate the Java source code, it will also need permissions for creating new directories as well as new files. Pre-existing files with the same name as a new file to be created will be overwritten.

Note that for the processing of XML, the Cider Compiler includes software developed by the Apache Software Foundation (`http://www.apache.org/`).

F Input Specification Grammar

This grammar specifies the format used to create CIDER specification files. The format is given using Extended Backus-Naur Form (EBNF) notation.

Characters

```
[1]   Char                ::=  (Any character)

[2]   NonNewlineChar      ::=  Char [p.60] - ‘\n’

[3]   Letter              ::=  [a-z] | [A-Z]

[4]   Digit               ::=  [0-9]

[5]   NameChar            ::=  Letter [p.60] | Digit [p.60] | ‘_’

[6]   S                   ::=  (‘ ’ | ‘\t’ | ‘\n’)+

[7]   OpS                 ::=  (‘ ’ | ‘\t’ | ‘\n’)*
```

Tokens

```
[8]   ProperName          ::=  Letter [p.60] (NameChar [p.60])*

[9]   Comment             ::=  ‘#’ (NonNewlineChar [p.60])*

[10]  QuotedString        ::=  ‘"’ ((NonNewlineChar [p.60] - ‘"’) | ‘\"’)* ‘"’ |
                               ‘'’ ((NonNewlineChar [p.60] - ‘'’) | ‘\'’)* ‘'’

[11]  FPNumber            ::=  (Digit [p.60] | ‘-’ | ‘.’  | ‘e’ | ‘E’)+
```

Data Types

```
[12]  Datatype            ::=  ‘int’ | ‘short’ | ‘long’ | ‘byte’ | ‘char’ |
                               ‘boolean’ | ‘float’ | ‘double’ | QualifiedClassName
                               [p.60] | ConstraintVariable [p.60]

[13]  QualifiedClassName  ::=  (NameChar [p.60] | ‘.’)+

[14]  ConstraintVariable  ::=  ‘@cv’ (‘[’ FPNumber [p.60] ‘,’ OpS [p.60] FPNumber
                               [p.60] ‘]’)?
```

Grammar File Contents

```
[15]  GrammarFile         ::=  GrammarName [p.60] S [p.60] SymbolDefs [p.61]
                               ProductionRules [p.61]

[16]  GrammarName         ::=  ‘grammar’ OpS [p.60] ‘=’ OpS [p.60] (‘"’
                               ProperName [p.60] ‘"’ | ‘'’ ProperName [p.60]
                               ‘'’) OpS [p.60] ‘;’
```

Specifying Symbols

[17]	SymbolDefs	::=	‘symbol_definitions’ S [p.60] ‘{’ S [p.60] (SymbolDefinition [p.61] S [p.60])+ ‘}’
[18]	SymbolDefinition	::=	ProperName [p.60] S [p.60] ‘(’ (‘terminal’ | ‘non-terminal’ | ‘start’ | ‘abstract’) ‘)’ (S [p.60] ‘[’ QualifiedClassName [p.60] ‘]’)? (S [p.60] ‘:’ OpS [p.60] ProperName [p.60])? (AttList [p.61])? OpS [p.60] ‘;’
[19]	AttList	::=	S [p.60] AttributeDef [p.61] (OpS [p.60] ‘,’ S [p.60] AttributeDef [p.61])*
[20]	AttributeDef	::=	ProperName [p.60] ‘:’ Datatype [p.60]

Specifying Production Rules

[21]	ProductionRules	::=	(S [p.60] Production [p.61])*
[22]	Production	::=	(ProperName [p.60])? ‘:production’ S [p.60] ‘{’ S [p.60] SymbolList [p.61] S [p.60] ‘::=’ S [p.60] SymbolList [p.61] S [p.60] ConstraintSet [p.61] S [p.60] AttAssignment [p.62] ’}’
[23]	SymbolList	::=	Symbol [p.61] (OpS [p.60] ‘,’ S [p.60] Symbol [p.61])*
[24]	Symbol	::=	ProperName [p.60] ‘:’ ProperName [p.60] ‘*’?

Specifying a Constraint Set

[25]	ConstraintSet	::=	‘where’ S [p.60] (‘true’ OpS [p.60] ‘;’ | ‘{’ S [p.60] (Constraint [p.61] S [p.60])+ ’}’)
[26]	Constraint	::=	SimpleComparison [p.61] | SolverConstraint [p.62] | ExistsConstraint [p.62]
[27]	SimpleComparison	::=	Argument [p.61] S [p.60] Relationship [p.62] S [p.60] Argument [p.61] OpS [p.60] ‘;’
[28]	Argument	::=	AttributeValue [p.61] | SymbolValue [p.61] | ConstantValue [p.61] | FunctionValue [p.62]
[29]	AttributeValue	::=	ProperName [p.60] ‘.’ ProperName [p.60]
[30]	SymbolValue	::=	ProperName [p.60]
[31]	ConstantValue	::=	‘true’ | ‘false’ | ‘null’ | QuotedString [p.60] | FPNumber [p.60] | ‘@java(’ QuotedString [p.60] ’)’

```
[32]  FunctionValue        ::=  QualifiedClassName [p.60] ‘.’  ProperName [p.60]
                                ‘(’ (ArgumentList [p.62])?  ‘)’

[33]  ArgumentList         ::=  Argument [p.61] (OpS [p.60] ‘,’ S [p.60] Argument
                                [p.61])*

[34]  Relationship         ::=  ‘==’ | ‘!=’ | ‘<=’ | ‘>=’ | ‘>’ | ‘<’ | ‘eq’ | ‘ne’

[35]  SolverConstraint     ::=  ‘@sc(’ FPNumber [p.60] OpS [p.60] ‘,’ S [p.60]
                                ConstraintString [p.62] ’)’ OpS [p.60] ‘;’

[36]  ConstraintString     ::=  ‘"’ SolverPolynomial [p.62] OpS [p.60] SolverRel
                                [p.62] OpS [p.60] FPNumber [p.60] ‘"’ | ‘'’
                                SolverPolynomial [p.62] OpS [p.60] SolverRel [p.62]
                                OpS [p.60] FPNumber [p.60] ‘'’

[37]  SolverPolynomial     ::=  PolynomialTerm [p.62] (OpS [p.60] ‘+’ OpS [p.60]
                                PolynomialTerm [p.62])?

[38]  PolynomialTerm       ::=  (FPNumber [p.60] | ‘-’)?  OpS [p.60] ‘[’
                                AttributeValue [p.61] ‘]’

[39]  SolverRel            ::=  ‘==’ | ‘<=’ | ‘>=’

[40]  ExistsConstraint     ::=  (‘exists’ | ‘not_exists’) S [p.60] Symbol [p.61] S
                                [p.60] ConstraintSet [p.61]
```

Assigning Attribute Values

```
[41]  AttAssignment        ::=  (InitAtts [p.62] S [p.60])?  (SetAtts [p.62] S
                                [p.60])?

[42]  InitAtts             ::=  ‘init’ OpS [p.60] ‘{’ S [p.60] InitAttribute [p.62]
                                S [p.60] ’}’

[43]  InitAttribute        ::=  AttributeValue [p.61] S [p.60] ‘:=’ S [p.60]
                                Argument [p.61] OpS [p.60] ‘;’

[44]  SetAtts              ::=  ‘set’ OpS [p.60] ‘{’ S [p.60] SetAttribute [p.62] S
                                [p.60] ’}’

[45]  SetAttribute         ::=  AttributeValue [p.61] S [p.60] ‘:=’ S [p.60]
                                (Argument [p.61] | ConstraintExp [p.62]) OpS [p.60]
                                ‘;’

[46]  ConstraintExp        ::=  ‘@ce(’ ExpressionString [p.62] ’)’

[47]  ExpressionString     ::=  ‘"’ SolverExp [p.63] ‘"’ | ‘'’ SolverExp [p.63]
                                ‘'’
```

[48]	SolverExp	::=	SolverPolynomial [p.62] | FPNumber [p.60] | SolverPolynomial [p.62] OpS [p.60] '+' OpS [p.60] FPNumber [p.60]

Transformations File Contents

[49]	TransformationsFile	::=	((BasicTrans [p.63] | Disjunction [p.63] | CompoundTrans [p.63]) S [p.60])*

Simple Transformations

[50]	BasicTrans	::=	ProperName [p.60] ':transformation' S [p.60] '{' S [p.60] (SymbolList [p.61] S [p.60])? ConstraintSet [p.61] S [p.60] (ActionList [p.63] S [p.60])? '}'
[51]	ActionList	::=	'apply' S [p.60] '{' S [p.60] (Action [p.63] S [p.60])* '}'
[52]	Action	::=	NewTerminalSymbol [p.63] | ApplyProduction [p.63] | SetAttValue [p.63]
[53]	NewTerminalSymbol	::=	'(nts)' S [p.60] Symbol [p.61] '(' ArgumentList [p.62] ')' OpS [p.60] ';'
[54]	ApplyProduction	::=	'(ap)' S [p.60] ProperName [p.60] (S [p.60] 'using(' MappingList [p.63] ')')? OpS [p.60] ';'
[55]	MappingList	::=	Mapping [p.63] (OpS [p.60] ',' S [p.60] Mapping [p.63])*
[56]	Mapping	::=	ProperName [p.60] OpS [p.60] ('->' | '<-') OpS [p.60] ProperName [p.60]
[57]	SetAttValue	::=	'(set)' S [p.60] InitAttribute [p.62]

Complex Transformations

[58]	Disjunction	::=	ProperName [p.60] ':disjunction' S [p.60] '{' S [p.60] DisjunctionList [p.63] S [p.60] '}'
[59]	DisjunctionList	::=	ProperName [p.60] (OpS [p.60] '|' OpS [p.60] ProperName [p.60])+ OpS [p.60] ';'
[60]	CompoundTrans	::=	ProperName [p.60] ':compound_transformation' S [p.60] '{' S [p.60] CompoundString [p.63] S [p.60] '}'
[61]	CompoundString	::=	CompoundTerm [p.64] OpS [p.60] ';'

[62] CompoundTerm ::= ProperName [p.60] '!'? | '(' OpS [p.60] CompoundTerm [p.64] OpS [p.60] ')' | CompoundTerm [p.64] OpS [p.60] '*' | CompoundTerm [p.64] OpS [p.60] '|' OpS [p.60] CompoundTerm [p.64] | CompoundTerm [p.64] S [p.60] CompoundTerm [p.64]

References

[1] R. Bardohl, M. Niemann, and M. Schwarze. GenGed – a development environment for visual languages. In *Applications of Graph Transformations with Industrial Relevance*, pages 233–240, 2000.

[2] B.T. VanDer Zanden et al. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):776–796, 2001.

[3] S.S. Chok and K. Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of the 11th ACM Symposium on User Interface Software and Technology*, 1998.

[4] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, New Jersey, 1999.

[5] V. Haarslev. A fully formalized theory for describing visual notations. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 261–292. Springer, New York, 1998.

[6] A.R. Jansen, K. Marriott, and B. Meyer. CIDER: A component-based toolkit for creating smart diagram environments. In Shu-Ching Chen, editor, *Proceedings of the Ninth International Conference on Distributed and Multimedia Systems*, pages 353–359. Knowledge Systems Institute, 2003.

[7] K. Marriott. Constraint multiset grammars. In *IEEE Symposium on Visual Languages*, pages 118–125, 1994.

[8] K. Marriott and S.S. Chok. Qoca: A constraint solving toolkit for interactive graphical applications. *Constraints*, 7(3):229–254, 2002.

[9] K. Marriott and B. Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4):374–402, 1997.

[10] K. Marriott, B. Meyer, and K. Wittenburg. A survey of visual language specification and recognition. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 5–85. Springer-Verlag, New York, 1998.

[11] B. Meyer, K. Marriott, and G. Allwein. Intelligent diagrammatic interfaces: State of the art. In P. Olivier, M. Anderson, and B. Meyer, editors, *Diagrammatic Representation and Reasoning*. Springer-Verlag, London, 2001.

[12] B. Meyer, H. Zweckstetter, L. Mandel, and Z. Gassmann. Automatic construction of intelligent diagrammatic environments. In *HCI'99: 8th Int. Conf. on Human-Computer Interaction*, Munich, August 1999.

[13] M. Minas and G. Viehstaedt. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *IEEE Workshop on Visual Languages*, pages 203–210, 1995.