Triangle: Language, Literature, Computation, n. 1, 2010
Publicacions Universitat Rovira i Virgili · ISSN: 2013-939X
https://revistes.urv.cat/index.php/triangle

Logic Programming for Linguistics: A short introduction to Prolog, and Logic Grammars with Constraints as an easy way to Syntax and Semantics

Henning Christiansen

CBIT Institute, Roskilde University, Denmark
http://www.ruc.dk/~henning/, henning@ruc.dk

Introduction

This article gives a short introduction on how to get started with logic programming in Prolog that does not require any previous programming experience. The presentation is aimed at students of linguistics, but it does not go deeper into linguistics than any student who has some ideas of what a computer is, can follow the text. I cannot, of course, cover all aspects of logic programming in this text, and so we give references to other sources with more details.

Students of linguistics must have a very good motivation to spend time on programming, and I show here how logic programming can be used for modelling different linguistic phenomena. When modelling language in this way, as opposed to using only paper and pencil, your models go live: you can run and test your models and you can use them as automatic language analyzers. This way you will get a better understanding of the dynamics of languages, and you can check whether you model expresses what you intend.

Based on Prolog, I also introduce Definite Clause Grammars which is integrated in most Prolog systems: You can write a grammar in a straightforward notation, perhaps include different syntactic, semantic and pragmatic features – and with no additional effort, you can use it as an automatic language analyzer.

I show also another important extension to Prolog, called Constraint Handling Rules, which boosts these grammars with capabilities for capturing semantics and pragmatics by abductive reasoning, in a way that I claim is considerably simpler than mainstream formalisms; this part is to a large extent based on my own research.

Hardcore linguists may object that these approaches are too simplistic – and they are right (of course, they are always right ;-) – but this simplicity, I will reply, provides exposure to linguistic phenomena in a clarified and distilled form which is difficult to obtain by other means.

Finally I apologize for any errors, omissions, misspellings and mistakes, which I'm sure there are plenty of, as this article has been produced in a very short time. I am glad to receive any comments and questions.

All example programs can be downloaded from the following website: http://www.ruc.dk/∼henning/LP-for-Linguists.

1 Prolog: Programming without programming

Prolog is one of the easiest programming languages to use for a beginner in programming: You only need to learn a few simple basic structures, and you can start on your own. Programs are given as plain text files which you can edit with any plain text editor.

1.1 Prolog, lesson 1: A program as a knowledge base of facts

The following is our first Prolog program; we will assume that it is kept in a file named `royal`.[1]

```
% Danish Royal Family

parent( margrethe, frederik).
parent( margrethe, joachim).
parent( henrik, frederik).
parent( henrik, joachim).
parent( mary, christian).
parent( mary, isabella).
parent( frederik, christian).
parent( frederik, isabella).
parent( alexandra, nicolai).
parent( alexandra, felix).
parent( joachim, nicolai).
parent( joachim, felix).
parent( marie, little_henrik).
parent( joachim, little_henrik).
```

The very first character in the program "%" indicates that the rest of that line is a comment. The rest of this program consists of *facts*, in this example listing the parental relationships for a part of the Danish royal family. The meaning of the program is not that the program should be executed from beginning to end, one instruction at a time, but it is to be understood as a *knowledge base.*

[1] In some operation systems, it will best to give the file name an extension, e.g. `royal.txt`. The extension "`.pl`" is also common, but this makes many systems believe that the file is a Perl program, which is something completely different.

I suppose this meaning becomes clear to you, simply by looking at the program. You will also observe that these facts are written in a fixed format; here `parent` is called a *predicate*, and the names that appear, such as `margrethe`, are examples of *constant symbols* or, for short, *constants*.[2] It is important to notice that each fact must be ended with a period ".".

We can run a program by asking *queries*. A query is a sort of question that the Prolog system tries to answer as good as it can. We will try some examples. The following shows a dialogue with a computer that has a version of Prolog installed; we assume that it is started by the command `prolog`, but this may vary; before you start, be sure to be in the directory that contains the program file `royal`. The following is a listing of the command window after a dialogue between a user and the Prolog system.

```
$ prolog
................
| ?- [royal].
% compiling directory/royal...
yes
| ?- parent(margrethe, frederik).
yes
| ?- parent(margrethe, obama).
no
| ?- parent(margrethe, juan_carlos).
no
| ?- parent(margrethe, X).
X = frederik ? ;
X = joachim ? ;
no
| ?- parent(X,felix).
X = alexandra ? ;
X = joachim ? ;
no
| ?- halt.
$
```

[2] Some books and manuals also use the term, an *atom*, which is a bit misleading since an atom is something different in the mathematical logic on which Prolog is based.

You cannot see who (user or computer) wrote which part of the text, but I will explain. The $ sign is the prompt from the computer's operating system, and the user starts Prolog by typing the command `prolog`. Where you see "`.....`", you will typically get a message saying which Prolog system and version you are using, but this is not interesting. The symbols "`| ?-`" are printed by the system, meaning that it expects input from the user. Our nice user first loads the program by writing "`[royal].`";[3] this is the general syntax for loading in programs; notice the terminating period, and that is the same for any query that you type following "`| ?-`", and you should also type end-of-line at the end. Now the system gives a response, saying that it has accepted the program. which is now ready in its memory.

Now we can start asking queries, and assume that our nice user types "`parent(margrethe, frederik)`". When typed in like this, you should understand this as a question "*Is it true that ...*"; here the system answers "`yes`", which means that it has found out that, this is indeed true according to the program.[4] In this case, it is easy for us to check that Prolog was right since the query matches a fact in the program. Let us try a more advanced query, "`parent(margrethe, juan_carlos)`". Here the system replies "`no`" meaning that the query cannot be shown to be true according to the program; we can easily check that this conclusion is correct.[5]

[3] If your computer requires files with an extension as in "`royal.txt`" or "`royal.pl`", it often works to load the file without writing the extension, `| ?- [royal].` If that does not work, you may need to write the extension as well, e.g., `| ?- ['royal.txt'].` Notice when you do this that single quotes are essential, otherwise Prolog gets confused by the period and emits a weird error messsage,

[4] You may notice the dubious usage of "true" and "truth"; what I mean here is not that something is true in the real world, but it is a logical consequence of the program. In fact, the computer has no coupling between the constant `margrethe` and a real living person, who happens to be Her Majesty, the Queen of Denmark. This meaning is reserved for humans, based on our intuition and knowledge about the world; if the program is wrong according to the real world, its answers will of course be wrong. The other way round, whether or not the program is correct, we can say that it defines a set of possible worlds, and something is stated to be true by Prolog only when it is true in all those possible worlds. However, this is too philosophical for us, so I shall leave it for now.

[5] Notice that Prolog considers anything to be false, as indicated by "`no`", that cannot be shown to be correct by the program. This is also problematic as something might be true in the real world even if it is not mentioned in the program. In fact, it is difficult to imagine a program that contains all the knowledge about

Now our nice user tries something really advanced, namely to use a *variable* in the query "`parent(margrethe, X).`"; notice that variables are indicated by initial capital letter whereas a constant starts with a small letter. The meaning of such a query is a request for "which values of the variable makes my query true". So when our nice user queries "`parent(margrethe, X).`", it means that she wants to know which people that have `margrethe` as a parent. As it appears, the systems tells the nice user that there are two possibilities, namely `X=frederik` and `X=joachim`; also this time we can compare with the program text, that this is indeed a sound conclusion.

You should be aware that after each answer, when Prolog states a question mark "`?`" as shown, it should be understood as "do you want another solution?". Here our nice user needs to type a semicolon "`;`" if she wants to confirm that she wants another solution; the final "`no`" should be taken as "no *more* answers". If the first answer is sufficient, simply type end-of-line after the question mark.

The next query also uses a variable, but in a different position, namely for asking who are the parents of `felix`. It is important to learn from these example, that in Prolog, no specific positions of a predicate should be thought of as specific for "input" and others specific for "output"; you can use them as you please.

Finally, the query "`halt.`" is a command to Prolog that we want to stop and return to the operating system.

1.2 Prolog, lesson 2: Using variables to combine information and writing rules

As I wrote above, using a variable in a query was a suggestion for the system to fill in constants, so that the query becomes true. In fact, we can use several variables in a query and also, the same variable can appear several times. Let us consider an example, and assume that our nice user is interested to know, who the queen's grandchildren are; obviously, this information is

the real world. So "false" in Prolog's terms does not mean that it is really false, but rather that the program does not contain information about it. So it might be more correct to have the system state "`I don't know`" rather than "`no`", but that is too difficult to say; "`no`" is easier, and as soon as you know what a "`no`" means, this should not be a problem. — I promised in the last footnote not to include any more philosophical discussion, so this is definitely the very last footnote of this kind.

embedded in the program, but not in an explicit way. She now starts the system again, loads the program, and tries the following query.

```
| ?- parent(margrethe,X), parent(X,Y).
```

This is asking for pairs of values of `X` and `Y` which makes the query true. In other words, `X` should be a child of `margrethe`, and `Y` should be the child of the aforementioned child, i.e., `Y` should be a grandchild of `margrethe`. Let us see Prolog's answers when our nice user types semicolons after each to get more.

```
X = frederik,
Y = christian ? ;
X = frederik,
Y = isabella ? ;
X = joachim,
Y = nicolai ? ;
X = joachim,
Y = felix ? ;
X = joachim,
Y = little_henrik ? ;
no
```

It may be a bit difficult to read when you see everything at the same time, but each answer is ended by the semicolon typed by the user. So for example, `isabella` is a grandchild of `margrethe` because `isabella` has `frederik` as a parent, and `frederik` has `margrethe` as a parent.

Our nice user may become a bit tired both from writing the complicated expressions, each time having to get the `X`s and `Y`s right, and from seeing `margrethe`'s. There is a remedy in Prolog to this namely the possibility of defining a *rule* as part of the program. Our nice user suggests this rule:

```
grandparent(X,Z):- parent(X, Y), parent(Y, Z).
```

She adds it to the program file and reads in the program file once again to test it. She now types in the following query and a number of semicolons to get the following answers.

```
| ?- grandparent(margrethe,X).
X = christian ? ;
X = isabella ? ;
X = nicolai ? ;
```

```
X = felix ? ;
X = little_henrik ? ;
no
```

As it appears, this is exactly what she wants. The meaning of the rule is straightforward: in order to evaluate `grandparent(X,Z)` for which the program has no facts, evaluate instead `parent(X, Y), parent(Y, Z).`, and return what was found for `X` and `Z`, thus saying nothing about the values of `Y` as it is completely local to the body of the clause.

Notice that we used the variable names freely as we liked. When using an `X` as the query, we do not have to consider whether the rule uses `X` for some other purpose; and if we have several rules in the program, their different uses of `X` do not get mixed up. So when, in the example above, we wrote `margrethe` in the query where the rule uses `X`, and we wrote also `X` in the query where the rule uses `Z`, they do not get mixed up. The system is clever enough to replace variables and values so everything works out the right way.

This finishes Prolog lesson 1 and 2, which is the core of Prolog and with which you can already write a lot of interesting programs.

1.3 Prolog, lesson 3: The rest of Prolog, with a focus on lists

Prolog includes a lot of other things, of which the most important are:

- a lot of standard built-in predicates so you do not need to write them yourself every time; any comprehensive Prolog textbook or manual will tell you about them,
- structures, so that we can use structural information in predicates, and not only constants such as "`margrethe`"; a special kind of structures is lists that I will show below as they are important for language processing,
- some devices which makes it possible for you to affect the way Prolog is searching in its knowledge base for rules and facts in order to answer the queries; this can give essential speed-up to large programs but I ignore all that in this article.

The following is an an example of a Prolog list.

```
[once, upon, a, time]
```

It includes four constants, and as you see, I anticipate the use of lists to represent text. Prolog gives us some notation to work with lists, as we show in the following program.

```
first(H, [H | _]).
rest(R, [_ | R]).
```

Firstly, the underline character is used as a so-called anonymous variable; it adds nothing conceptually new to what we have already seen and is not restricted to lists. It can be used for a variable that we only use once in a rule (so there is no reason to give it an explicit name (think!)), which means that we do not care about its actual value. The vertical bar inside the list brackets is a special notation for lists, and separates the first element from the list of remaining elements. So for example, when `[once, upon, a, time]` is matched with `[A|B]`, it will lead to `A=once` and `B=[upon, a, time]`. Here are some queries to the program above together with its answers; the program is in a file called `firstlast`.

```
$ prolog
| ?- [firstlast].
% compiling directory/firstlast...
yes
| ?- first(F, [once, upon, a, time]).
F = once ?
yes
| ?- rest(R,  [once, upon, a, time]).
R = [upon,a,time] ?
yes
| ?- halt.
$
```

I will not spend more time on this example, but instead show a linguistically inspired example. I will later introduce a grammar notation which makes it easier to write, but now our point is to illustrate the use of lists for language. The following program (file `sheeps`) uses Prolog and its list notation to define a grammar for sheeps' utterances.

```
sheeptalk([]).
sheeptalk([M|Ms]):- sheepsound(M), sheeptalk(Ms).
sheepsound(mah).
sheepsound(meeeeeh).
```

The first rule tells that sheep can keep quiet when necessary, `[]` is a notation for the empty list. The second rule explains in a recursive way that `sheeptalk` consists of `sheepsounds`. In practice this means that it will clip off one atom at a time and check if it is a `sheepsound`, i.e., `mah` or `meeeeeh`. The following shows it at work.

```
$ prolog
................
| ?- [sheeps].
% compiling directory/sheeps...
yes
| ?- sheeptalk([mah,mah,meeeeeh]).
yes
| ?- sheeptalk([mah,mah,mouuuuuuuuuuuuuuuh]).
no
| ?- halt.
$
```

This is the essence of language analysis in Prolog; notice that you can think of such a program as a grammar, and that Prolog can automatically use it as an analyzer. And with a bit of imagination, you may be able to see that we can extend this with different predicates for nouns, verbs, adjectives, etc., and that an elaborate set of rules can express how some natural sentences may look like. However, in the next section, I will introduce a special grammar notation that most Prolog systems can use.

1.4 More reading

There are several good books that introduce to and go in depth with Prolog; for computer science students, I have good experience of using Bratko's book [3], but the first half of the book is also fairly accessible to other people. The online, and now also paperback, book [2] may be easier to access for linguists. I will also refer to my own course notes [10] which are biased towards applications in artificial intelligence, including computational linguistics, and databases; if you skip the very few mathematical formulas that appear occasionally, it can give you an easily read (hmmm, well, fairy easily read) introduction to these areas. The notes have the advantage that they also introduce Constraint Handling Rules, which we apply to semantic-pragmatic analysis below.

Prolog was originally developed by a research group in Marseilles led by Alain Colmerauer in the 1970s, and spreading of ot was strongly promoted by D.H.D. Warren's first efficient implementation of Prolog [36] and R.A. Kowalski's book from 1979 [29]; since 1982, there have been annual conferences, ICLP, International Conference on Logic Programming.

2 Definite Clause Grammars

Now you have understood the basic mechanics of Prolog, I will introduce you to its grammar notation, called *Definite Clause Grammars* or *DCGs* among friends, by means of an example. You can write such rules directly in your Prolog program files, and you can mix Prolog and DCGs whenever you wish.

2.1 DCG, lesson 1: The basic grammar notation and syntax analysis

The `sheeps` program shown above can be written alternatively using grammar rules as follows; we assume it is contained in a file `sheepsGrammar`.

```
sheeptalk--> [].
sheeptalk--> sheepsound, sheeptalk.
sheepsound--> [mah].
sheepsound--> [meeeeeh].
```

You can see that we avoid explicitly clipping off constants one at a time, and we do not have to write list arguments explicitly. In a grammar, we may use *nonterminal (symbol)s* such as `sheeptalk`, and *terminal symbols* that are written in square brackets (i.e., the list notation re-used). In fact Prolog will translate, behind you back, a grammar such as `sheepsGrammar` into a Prolog program that resembles the `sheeps` Prolog program that I showed above. To query a grammar, i.e., to use it to analyse text, we need to use a special built-in predicate called `phrase`. It is shown at work below:

```
$ prolog
................
| ?- [sheepsGrammar].
% compiling directory/sheepsGrammar...
| ?- phrase(sheeptalk,[mah,mah,meeeeeh]).
yes
```

```
| ?- phrase(sheeptalk,[mah,mah,mouuuuuuuuuuuuuuuh]).
no
| ?- halt.
$
```

As you can see, DCGs provide a formal grammar notation, and you can use the Prolog system to tests examples to convince yourself that the grammar actually expresses what you have in mind. I claim that this is a very good reason for students of linguistics to work with these tools.

2.2 DCG, lesson 2: Adding features

A grammar can do more than just say `yes` and `no`, because we can add all kinds of features to the nonterminals, in a very similar way to how we used arguments for the predicates.

I will use a simplistic extension to the `sheepsGrammar` to illustrate this. I will consider how much grass a sheep needs to eat in order to perform a given speech; let us assume that a sheep needs one lump of grass to say `mah` and three lumps to say `meeeeeh`. For each syntactic phrase, we attach a feature that counts the total number of lumps for that phrase. This can be expressed as follows; you should notice the following details: the curly brackets `{···}` inside a grammar rule indicate a piece of Prolog code that should be interpreted whenever the given rule applies, and secondly Prolog's strange way of doing arithmetic by the "`is`" construction used below in order to perform an addition. Let the file `sheepsGrammarGrass` contain the following grammar.

```
sheeptalk(0)-->[].
sheeptalk(C)--> sheepsound(C1), sheeptalk(C2), {C is C1+C2}.
sheepsound(1)--> [mah].
sheepsound(3)--> [meeeeeh].
```

It works as follows.

```
$ prolog
................
| ?- [sheepsGrammarGrass].
% compiling directory/sheepsGrammar...
| ?- phrase(sheeptalk(C),[mah,mah,meeeeeh]).
C = 5 ?
```

```
yes
| ?- halt.
$
```

Finally I show a more interesting grammar for a subset of English; here I add a feature for number whenever it is relevant, and express the constraint that the number for noun phrase must match the number for the following verb phrase. Notice that number (indicated by variables called "N") can assume the values `plus` and `sing`. We write the grammar in the text file called `english` as follows.

```
s --> np(N), v(N), np(_).
np(N) --> noun(N).
np(plur) --> noun(_), [and], np(_).
noun(sing) --> [joachim].
noun(sing) --> [alexandra].
noun(sing) --> [marie].
noun(plur) --> [dogs].
v(sing) --> [likes].
v(plur) --> [like].
```

The following queries show it at work; I suggest that you inspect each query in detail and understand exactly why it answers as it does.

```
$ prolog
................
| ?- [english].
% compiling directory/english...
| ?- phrase(s, [joachim,likes,dogs]).
yes
| ?- phrase(s, [joachim,like,dogs]).
no
| ?- phrase(s, [marie,and,alexandra,likes,joachim]).
no
| ?- phrase(s, [marie,and,alexandra,like,joachim]).
yes
| ?- halt.
$
```

You can also extend your grammar with structures that represent syntax trees, so that when you analyze a sentence, you get as a result the tree that

represents the phrase structure of that sentence. It is straightforward to do so, and you can do it yourself, provided that you find a textbook or good course notes and read about structures in Prolog.

2.3 More reading

Definite clause grammars (DCG) were first presented in 1975 by A. Colmerauer [18] under the name of *grammaires de métamorphose*, and they got their final shape and name as DCGs in 1980 [31]. Any good Prolog textbook will have a section on Definite Clause Grammars, and they are included in virtually all available Prolog systems.

3 A brief introduction to Constraint Handling Rules, CHR, and their application for abductive reasoning

The term abduction usually refers to a kind of criminal act, quite different from the specific meaning that what I use it for here, and abductive reasoning sounds weird to most people.

I first give an introduction to the topic taken from [10], and then I introduce the language of Constraint Handling Rules by means of a few examples of how they can be used for adding abductive reasoning to Prolog. Then, I combine this with the grammar notation introduced above.

You may find the name and term "constraints" a bit confusing; this is a consequence of the application that CHR was originally designed for, which we discuss briefly in section 3.4 below.

Most applications of abductions, including the methods I introduce below, are used for diagnosis and planning; I will not go into such examples here, but you may try to think about the similarities between language interpretation and diagnosis.

3.1 Deduction, abduction, and induction in logic programming

The philosopher C.S.Peirce (1839–1914) is considered a pioneer in the understanding of human reasoning, especially in the specific context of scientific discovery. His work is often cited in computer science literature but probably only a few computer scientists have read Peirce's original work. I recommend [21] as an overview of Peirce's influence seen from the perspective of computer science.

Peirce postulated three principles as *the* fundamental ones:

- **Deduction**, reasoning within the knowledge we have already, i.e., from those facts we know and those rules and regularities of the world that we are familiar with. E.g., reasoning from causes to effects:
  "If you make a fire in the living room, you will burn down the house."
- **Induction**, finding general rules from the regularities that we have experienced in the facts that we know; these rules can be used later for prediction:
  "Every time I made a fire in my living room, the house burnt down, aha, ... the next time I make a fire in my living room, the house will burn down too".
- **Abduction**, reasoning from observed results to the basic facts from which they follow. Quite often it means from an observed effect to produce a qualified guess for a possible cause:
  "The house burnt down. Perhaps my cousin made a fire in the living room again."

In fact, Peirce had alternative theories and definitions of abduction and induction; I have adopted the so-called syllogistic version, cf. [21]. I can replicate the three in logic programming terms:

- A Prolog system is a purely deductive engine. It takes a program of rules and facts, and it can calculate or check the logical consequences of that program.
- Induction is difficult; methods for so-called inductive logic programming (ILP) have been developed, and by means of a lot of statistics and other complicated machinery, they synthesize rules from collections of "facts" and "observations". I can refer to [4][6] for an overview of different applications. Inductive logic programming has been successfully applied for molecular biology concerned with protein molecule shapes and human genealogy. See [30] for an in-depth treatment of ILP methods.
- Abductive logic programming; roughly means from a claim of goal that is required to be true (i.e., being a consequence of the program), to extend to program with facts so that the goal becomes true. See [27] for an overview. Abduction has many applications; I may mention planning (e.g., the goal is "successful project ended" and the facts to be derived are the detailed steps of a plan to achieve that goal), diagnosis (goal

[6] A bit old; if you are interested, you should search for more recent overview papers and consult proceedings of the recent ILP conferences; see http://www.informatik.uni-trier.de/~ley/db/conf/ilp/index.html.

is observed symptoms, the facts to be derived comprise the diagnosis, i.e., which specific components of the organism or technical system that malfunction). An important area for abduction is language processing, especially discourse analysis (the discourse represents the observations, the facts to be derived constitute an interpretation of that discourse). We will look into some of these in more detail below and give references.

However, we should be aware that while deduction is a logically sound way of reasoning, this is generally not the case for abduction and induction. Let me make a simple analysis for abduction. Assume a logical knowledge base $\{a \rightarrow c, b \rightarrow c\}$ where the arrow means logical implication. If we know c, an abductive argument may propose that a is the case. However, this is not necessarily true as it might that b is the case and not a. Or it could even be the case that none of a and b are the case, and that there is another and unknown explanation for c. Abduction is often described as reasoning to the best explanation. i.e., best with respect to the knowledge we have available.

3.2 Introducing Constraint Handling Rules by examples of abduction

Constraint Handling Rules [22], CHR, is a declarative, rule-based language for writing constraint solvers and is now included as an extension of several versions of Prolog. Operationally and implementation-wise, CHR extends Prolog with a constraint store, and the rules of a CHR program serve as rewriting rules over constraint stores. CHR is declarative in the sense that its rules can be understood as logical formulas. I show first a program in Prolog that does not use CHR and we analyze its deficiencies; it is given as the file `happy1`.

```
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

It is supposed to describe how someone can become happy, which, however, does not fit exactly with Prolog's mode of working, as we will see. We ask now the following query with the intension of finding out how someone with the name `henning` can be happy.

```
| ?- happy(henning).
! Existence error in user:rich/1
! procedure user:rich/1 does not exist
! goal:  user:rich(henning)
```

It goes wrong because Prolog needs to investigate calls to `rich`, which is not defined by any facts or rules. By giving a suitable command to Prolog (which I don't show here), we can get rid of the error message, so that a call to a predicate with zero facts and rules always fails (as opposed to crashing), which is more in accordance with a logical meaning of the Prolog program.[7] In this case we would get the answer `no` instead since the predicates `rich`, `professor` and `has` are all empty, but this is still not satisfactory according to our intension with the query.

What we wanted to achieve, was one or more explanations of how we could get the conclusion `happy(henning)`, and to do this, we must make a part of the program dynamic in the sense that the system should be able to add facts to see if that made the goal succeed. Now you may see the relationship with abductive reasoning, which, as I have shown, is beyond plain Prolog's capabilities.

We can now use CHR to declare the predicates `rich`, `professor` and `has` as *constraints*, in the sense that they are now governed by the CHR system. We do this in the next version of the program, `happy2`; the first line is necessary for making CHR available.

```
:- use_module(library(chr)).
:- chr_constraint rich/1, prof/1, has/2.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

Having these predicates declared as constraints will have the effect that they 1) are not unknown anymore, and 2) whenever they are called, the system will add the calls to its constraint store. At the end, the collected constraint store is printed out as part of the answer. We test the `happy2` program and get the following results.[8]

[7] There is a very good reason, though, why Prolog as default emits an error message rather than silently failing if a non-existing predicate is called. Can you imagine, if you have a very big program over several thousand lines, and you have misspelled one occurrence of a predicate; the error message will help you to locate the error while a failure would make it almost impossible to detect if, e.g., the program simply answers "`no`".

[8] **Important note concerning SWI Prolog:** Some older versions do not print out the constraint store when the program finishes; if you experience this problem, check the manual for the version you are using, to find the command that makes it print the constraint store. Otherwise, there is not much fun in using CHR for abduction!

```
| ?- happy(henning).
rich(henning) ? ;
professor(henning),
has(henning,nice_students) ? ;
no
```

As you can see, the two alternative answers say that there are two ways that `happy(henning)` can be true, namely if either the constraint store contains `rich(henning)` or, alternatively, `professor(henning)` and `has(henning, nice_students)`.

I will relate these answers to abductive reasoning as follows:

If we forget everything about CHR and type in, say, `rich(henning)` *as a part of the program, then* `happy(henning)` *will succeed, i.e., answer "*`yes`*".*

However, we can improve this program even further and make it better to reflect the real world. It is a fact that university professors are much lower paid than people in the industry with less education, and we always complain about this. We should somehow express this in our program, and here the rules of CHR come in handy. Rules in CHR operate on the constraint store, and a rule *fires*, whenever the total set of constraints in the store makes it possible for that rule to apply. We show this in an improved version of the program, `happy3`.

```
:- use_module(library(chr)).
:- chr_constraint rich/1, prof/1, has/2.
prof(X), rich(X) ==> fail.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

The construction written with "`==>`" is a CHR rule of the kind called a *propagation rule.* The meaning is that when its head (the left hand side) matches constraints in the store, the body (right hand side) is executed; in the example the body amounts to "`fail`" which will cause the system to try another branch. Now let us see how this program works; notice that the program does not know that `professor(henning)`, so we need to state this as part of the query to get the right answers.

```
| ?- happy(henning), professor(henning).
professor(henning),
professor(henning),
has(henning,nice_students) ? ;
```

```
no
```

Here we get only one answer, namely that `professor(henning)` and `has(henning,nice_students)`. The alternative postulation a professor to be rich is removed due to the CHR rule. You may notice that the constraint `professor(henning)` is repeated in the answer; this is due to some technical reasons that I will not spend time on explaining, it does not mean any thing.

This last example basically shows the part of CHR that you need to know how to use it for abductive language interpretation, as shown below.

Finally, I will comment on some terminological confusion that appears because this way of doing, involves usages from different areas.

- *"Constraint"* refers in CHR context to predicates that have been declared in as such, and that are treated by the system in a special way. We use CHR constraints here for what in the tradition of abductive reasoning is called *abducibles.*
- *"Integrity constraint"* refers in database theory and in abductive reasoning not to the simple piece of information, but to general knowledge about the world, about what is possible and what is not. Above, we used a CHR rule to describe an integrity constraint.
- Finally, as you have noticed, Prolog rules and CHR rules are something completely different, so referring to *"a rule"* may be ambiguous.

3.3 Details of CHR

Most of my readers may skip this subsection as you can make interesting linguistic applications, by generalizing from the examples above. The rest of this section is taken verbatim from [10], and may contain a few terms that you may be unfamiliar with.

CHR takes over the basic syntactic and semantic notions from Prolog and extends them with its specific kinds of rules. The execution of CHR programs is based on a *constraint store*, and the effect of applying a rule is to change the effect of the store. For a program written in a combination of Prolog and CHR, the system switches between two tow. When a Prolog goal is called, it is executed in the usual top-down (or goal-directed) way, and when a Prolog rule calls a CHR constraint, this will be added to the constraint store — then the CHR rules apply as far as possible, and control then returns to the next Prolog goal.

Technically speaking, CHR constraints are first-order atoms whose predicates are designated constraint predicates, and a constraint store is a set of such constraints, possible including variables that are understood existentially quantified at the outermost level. A constraint solver is defined in terms of rules which can be of the following two kinds.

$$\text{Simplification rules: } c_1,\ldots,c_n \;\texttt{<=>}\; \mathit{Guard} \mid c_{n+1},\ldots,c_m$$
$$\text{Propagation rules: } c_1,\ldots,c_n \;\texttt{==>}\; \mathit{Guard} \mid c_{n+1},\ldots,c_m$$

The c's are atoms that represent constraints, possibly with variables, and a simplification rule works by replacing in the constraint store, a possible set of constraints that matches the pattern given by the *head* $c_1,\ldots c_n$ by the constraints given by the *body* $c_{n+1},\ldots,c_m$, although only if the condition given by *Guard* holds. A propagation rule executes in a similar way but without removing the head constraints from the store. What is to the left of the arrow symbols is called the *head*[9] and what is to the right of the guard the *body*. The declarative semantics is hinted by the applied arrow symbols (bi-implication, resp., implication formulas, with variables assumed to be universally quantified) and it can be shown that the indicated procedural semantics agrees with this. This is CHR explained in a nutshell.

CHR provides a third kind of rules, called *simpagation rules*, which can be thought of as a combination of the two or, alternatively, as an abbrevation for a specific form of simplification rules.

$$\text{Simpagation rules: } c_1,\ldots,c_i \setminus c_{i+1},\ldots c_n \;\texttt{<=>}\; \mathit{Guard} \mid c_{n+1},\ldots,c_m$$
$$\text{which can be thought of as: } c_1,\ldots,c_n \;\texttt{<=>}\; \mathit{Guard} \mid c_1,\ldots,c_i,c_{n+1},\ldots,c_m$$

In other words, when applied, $c_1,\ldots,c_i$ stays in the constraint store and $c_{i+1},\ldots,c_n$ are removed.

In practice, the body of CHR rules can include any executable Prolog expression including various control structures and calls to Prolog predicates. Similarly, Prolog rules and queries can make calls to constraints which, then, may activate the CHR rules.

The guards can be any combination of predicates (built-in or defined by the programmer) that test the variables in the head, but in general guards should not change the values of these variables or call other constraints; in these cases, the semantics gets complicated, see references given above if

[9] Some authors call each constraint to the left of the arrow a head, and with that terminology, CHR has multi-headed rules.

you are interested in the details. Finally, guards can be left out together with the vertical bar, corresponding to a guard that always evaluates to true.

The following example of a CHR program is adapted from the reference manual [33]; from a knowledge representation point of view it may seem a bit strange, but it shows the main ideas. It defines a a little constraint solver for a single constraint `leq` with the intuitive meaning of less-than-or-equal. The predicate is declared to be an infix operator to enhance reading, but this is not necessary (`X leq Y` could be written equivalently as `leq(X,Y)`).

```
:- use_module(library(chr)).
handler leq_handler.
constraints leq/2.
:- op(500, xfx, leq).

X leq Y <=> X=Y | true.
X leq Y , Y leq X <=> X=Y.
X leq Y \ X leq Y <=> true.
X leq Y , Y leq Z ==> X leq Z.
```

The first line loads the CHR library which makes the syntax and facilities used in the file available. The `handler` directive is not very interesting but is required. Next, the constraint predicates are declared as such (here only one such predicate) and this informs the Prolog system that occurrences of these predicates should be treated in a special way.

The program consists of four rules, one propagation, two simplifications, and one simpagation. The first simplification describes the transitivity of the `leq` constraints. If, for example, the constraints `a leq b` and `b leq c` are called, this rule can fire and will produce a new constraint `a leq c` (which in turn may activate other rules).

The second rule is a simplification rule which will remove the two constraints and unify the arguments. Intuitively, the rule says that if some `X` is less than or equal to some `Y` and the reverse also holds, then they should be considered equal (antisymmetry). With constraint store {`a leq Z`, `Z leq a`}, the rule can apply, by removing the two constraints and unifying variable `Z` with the constant symbols `a`.

Consider a slightly different example, the constraint store {`a leq b`, `b leq a`}. Again, the rule can apply, by removing the two constraints from the store and calling `a=b`. This will fail as `a` and `b` are two different constant symbols.

Notice that CHR is a so-called *committed choice* language in the sense that when a rule has been called, a failure as exemplified above will not result in backtracking. I.e., in the example, the observed failure will **not** add {`a leq b, b leq a`} back to the constraint store so other and perhaps more successful rules may be tried out. However, when CHR is combined with Prolog, a failure such as the one shown will cause Prolog to backtrack, i.e., it will undo the addition of the last of the two, say `b leq a`, and go back to the most recent choice point.

The simplification rule `X leq Y <=> X=Y | true` will remove any `leq` constraint from the store with two identical arguments. This illustrates a fundamental difference between Prolog and CHR. Where Prolog uses unification when one of its rules is applied to some goal, CHR uses so-called matching. This means that the mentioned rule will apply to `a leq a` but not to `a leq Z`. In contrast, the application of Prolog rule `p(X,X):-···` to `p(a,Z)` will result in `a=Z` before the body is entered.

The third rule in the program above is a simpagation rule `X leq Y \ X leq Y <=> true` which serves the purpose of removing duplicate constraints from the store.

We consider the following query and see how the constraint store changes.

```
?- C leq A, B leq C, A leq B.
```

Calling the first constraint triggers no rule and we get the constraint store {`C leq A`}. Calling the next one will trigger the transitivity rule (the last rule), and we get {`C leq A, B leq C, B leq A`}. The last call in the query will trigger a sequence of events. When `A leq B` is added to the constraint store, it reacts, so to speak, with `B leq A` and the second rule applies, removing the two but resulting in the unification of `A` and `B`; for the sake of clarity, let us call the common variable, which is referred to by both `A` and `B`, `V1`. Now the constraint store is {`C leq V1, V1 leq C`}. The same rule can apply once again, unifying `C` and `V1`, so that the result returned for the query is the empty constraint store and the bindings `A=B=C`.

In general, when a query is given to a CHR program (or a program written in the combined language of CHR plus Prolog), the system will print out the final constraint store together with Prolog's normal answer substitution. An alternative solution can be asked for as in traditional Prolog by typing a semicolon.

I end the presentation of CHR by showing a few simple examples taken from the CHR web site [5]. This program by Thom Frühwirth evaluates the greatest common divisor of positive numbers.

```
:- use_module( library(chr)).
handler gcd.
constraints gcd/1.

gcd(0) <=> true.
gcd(N) \ gcd(M) <=> N=<M | L is M-N, gcd(L).
```

Here are a few test queries.

```
?- gcd(2),gcd(3).
?- X is 37*11*11*7*3, Y is 11*7*5*3, Z is 37*11*5, gcd(X),
   gcd(Y), gcd(Z).
```

The following program generates the prime numbers between 1 and n when given the query `?- primes(`n`).` It was written by Thom Frühwirth and adapted by Christian Holzbaur.

```
:- use_module(library(chr)).
handler primes.
constraints primes/1, prime/1.

primes(1) <=> true.
primes(N) <=> N>1 | M is N-1, prime(N), primes(M).
prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

3.4 More reading

Constraint Handling Rules (CHR) were developed by T. Frühwirth from around 1992, first publication [24], in order to have a declarative language for writing constraint solvers for, e.g., working with arithmetic in logic programming. Later, it turned out that CHR was suited to a much wider class of applications as illustrated in the present article. The use of CHR for abductive reasoning was discovered by S. Abdennadher and myself in 2000 [1] and later the ideas have been refined in my own work, largely in an inspiring collaboration with Veronica Dahl, see, e.g., [6, 11–15].

A recent book [23] gives a thorough, mainly theoretical treatment of all aspects of CHR, and the collection [32] gives an overview of recent applications and developments concerning CHR. See also [22,25] for good overview papers and the CHR website
http://www.cs.kuleuven.be/∼dtai/projects/CHR.
Since 2004, there have been annual workshops on CHR.

4 Language interpretation as abduction in Prolog+CHR

Now we have all the tools for doing abductive language interpretation: We have the DCG grammar notation for the syntax, and I will show how CHR can take care of a large portion of the semantic-pragmatic analysis. In fact, it is interesting to see how the use of abduction tends to remove the borderline between semantics and pragmatics.

4.1 Introducing abductive interpretation by examples

The following example was developed when I gave a talk for students at GRLMC in Tarragona, so that sets the context for the example. It may be possible that some people attend the talk while others are away; furthermore, we will be interested in who is able to see whom. Note that the example is not always perfect from an intuitive point of view, but its shows the method.

We make a first suggestion for a grammar that uses CHR to extract (a selected part of) the meaning of a given discourse. In this version, in file `discourse1`, we do not include any CHR rules. Notice that instead of having a general rule for sentences, we have specialized rules for the different sort of sentences that we want to analyze. This is not essential, but made in order to simplify this example; for larger applications, it may be advisable to use a more homogeneous format.

```
:- use_module(library(chr)).
:- chr_constraint at/2, sees/2.
story --> [] ; s, ['.'], story.
s --> np(X), [sees], np(Y), {sees(X,Y)}.
s --> np(X), [is,at], np(E), {at(E,X)}.
s --> np(X), [is,on,vacation], {at(vacation,X)}.
np(pedro)  --> [pedro].
```

```
np(maria)  --> [maria].
np(loli)   --> [loli].
np(grlmc)  --> [grlmc].
np(hennings_talk) --> [hennings,talk].
np(vacation) --> [vacation].
```

Let us show a few examples of using this grammar for language analysis. The following query analyzes a very simple sentence and represents its meaning as a CHR constraint.

```
| ?- phrase(story, [pedro,is,at,grlmc,'.']).
at(grlmc,pedro) ? ;
no
```

Let us try another, longer discourse:

```
phrase(story, [pedro,sees,maria,'.', pedro,sees,loli,'.',
               pedro,is,at,grlmc,'.', maria,is,at,hennings,
               talk,'.',loli,is,on,vacation,'.']).
at(vacation,loli),
at(hennings_talk,maria),
at(grlmc,pedro),
sees(pedro,loli),
sees(pedro,maria) ? ;
no
```

It appears that each sentence is "translated" into a formal form, but there is not much semantic-pragmatic processing involved. So let us add a few CHR rules to express a bit of simple every-day knowledge. The first rule says that if someone is at GRLMC, he or she is also in Tarragona; the next one says that anyone can only be in one location (it uses a so-called simpagation rule which removes that last of the matched `in` constraints, so that we avoid duplicate constraints). Next we express that if someone is at my talk, he or she is also at GRLMC, and finally, if someone is on vacation, he or she is not in Tarragona.[10]

[10] The `diff` constraint is a device that ensures that two items need to be different for the rest of the discourse. You may use instead Prolog's built-in `dif` (one `f`) instead, but my handcrafted version gives more readable output. It can be defined as follows; you do not need to read this; I include it for completeness only and to indicate that I have not hidden any code under the carpet to get it to work.

```
at(grlmc,X) ==> in(tarragona,X).
in(Loc1,X) \ in(Loc2,X) <=> Loc1=Loc2.
at(hennings_talk,X) ==> at(grlmc,X).
at(vacation,X) ==> in(Loc,X), diff(Loc,tarragona).
```

During an analysis of the discourse, these rules will fire as soon as they can and do some simple reasoning on the constraint store as the analysis proceeds. The program `discourse2` also contains these rules.

```
| ?- phrase(story, [pedro,sees,maria,'.', pedro,sees,loli,
                    '.',pedro,is,at,grlmc,'.', maria,is,at,
                    hennings,talk, '.',loli,is,on,vacation,'.']).
at(vacation,loli),
at(grlmc,maria),
at(hennings_talk,maria),
at(grlmc,pedro),
in(_A,loli),
in(tarragona,maria),
in(tarragona,pedro),
sees(pedro,loli),
sees(pedro,maria),
diff(_A,tarragona) ? ;
no
```

As it appears, the meaning extracted from the discourse now also includes for each person, in which place he or she is. Note that `loli` is in some place, referred to be a variable written by the system as "`_A`"; we do not know where this place is, except that it is not `tarragona`.

We will now make one last extension, `discourse3`, of the program to indicate who can see whom. If you are able to see someone then you are both in the same place, e.g., Tarragona, *or* you contact the person using skype. The following CHR rule needs a bit of explanation. The semicolon in the body signifies a logical "or" so that the system will try out both possibilities if asked for more answer or if the first alternative leads to a

```
:- chr_constraint diff/2.
diff(X,X) <=> fail.
diff(A,B) \ diff(A,B) <=> true.
diff(A,B) \ diff(B,A) <=> true.
diff(A,B) <=> ?=(A,B) | true.
```

failure. Secondly, you should ignore the irrelevant "`true |`" in the rule: there is a design bug in the CHR syntax so that when you use a semicolon in the body, you need to write it like this; there is no reason to make any effort to understand why.

```
see(X,Y) ==> true | (in(L,X), in(L,Y)  ;
in(Lx,X), in(Ly,Y), diff(Lx,Ly), skypes(X,Y)).
```

Let us try the usual query again with the program that contains all the rules shown so far.

```
| ?- phrase(story, [pedro,sees,maria,'.', pedro,sees,loli,
                '.',pedro,is,at,grlmc,'.', maria,is,at,
                hennings,talk, '.',loli,is,on,vacation,'.']).
at(vacation,loli),
at(grlmc,maria),
at(hennings_talk,maria),
at(grlmc,pedro),
in(_A,loli),
in(tarragona,maria),
in(tarragona,pedro),
see(pedro,loli),
see(pedro,maria),
skypes(pedro,loli),
diff(tarragona,_A) ? ;
no
```

We notice that the only answer is one in which Pedro sees Loli via skype, since the other option that they are in the same place is not possible: Pedro is in Tarragona and Loli is somewhere which is not Tarragona. This example has illustrated how the CHR rules can process the bits of information generated for each sentence and form it in into a knowledge base, that also contains knowledge that is not expressed directly in the discourse, but is somehow necessary for the discourse to be made.

There are still a few imperfections in this grammar, for example that the `sees` relationship is not symmetric, but we would expect that if *A* sees *B* then *B* also sees *A*; this is easy to repair (when you are familiar with CHR), but there is no reason to spend more time on this here.

4.2 More reading

The principle of seeing language interpretation as abduction was first formulated in [26], which is a highly referenced paper from 1993. Abduction in logic programming was studied from around 1990 or before with [28] as a central reference; see the following overview papers [20, 27]. The use of abduction implemented with CHR starts around 2000 with my own work; the first references are [6, 7]. Later I developed these ideas together with Veronica Dahl, which led to the combined use of DCG and CHR as demonstrated above. The work with Veronica also resulted in the Hyprolog system which is briefly described next.

In [16, 17], we have developed a realistic example of a CHR based grammar, which reads so-called use cases and produces UML diagrams. Use cases are used for the sort of system analysis that is made for the development of complex computerized systems that are typically used in large and complex organizations; use cases are small stories about what goes in the organization. A UML diagram, on the other hand, is a graphical representation of which classes of objects appear and their mutual relationships.

5 One step further: Hyprolog

Hyprolog is a system thought out by Veronica Dahl and myself, which puts an additional set of facilities on top of Prolog+DCG+CHR. The syntax for declaring abducible predicates is different (in Hyprolog, we call them `abducibles` rather than `chr_constraints`), and a few more aspects of abduction not described here are supported. Most notably, Hyprolog includes so-called assumptions that work very much like abducibles, but they also reflect the time which is implicit in a discourse — some things are said before and after certain other things — and they have explicit creation and applications.

I will not explain Hyprolog in detail, but you can refer to the articles [13, 14] and the Hyprolog User's Guide which is available at [9] together with source code and examples. First we describe these new devices, assumptions, and then we sketch a larger example available from [9].

5.1 Assumptions: Like abduction but with time

The text in this subsection is taken from [14], written together with Veronica Dahl.

Assumptive logic programs [19] are logic programs augmented with a) linear, intuitionistic and timeless implications scoped over the current continuation, and b) implicit multiple accumulators, particularly useful to make the input and output strings invisible when a program describes a grammar (in which case we talk of Assumption Grammars [19]). More precisely, we use the kind of linear implications called *affine* implications, in which assumptions can be consumed at most once, rather than exactly once as in linear logic. Although intuitively easy to grasp and to use, the formal semantics of assumptions is relatively complicated, basically proof theoretic and based on linear logic [19, 34, 35]. Here we use a more recent and homogeneous syntax for assumptions introduced in [8]; we do not consider accumulators, and we note that Assumption Grammars can be obtained by applying the operators below within a DCG.

`+h(a)`	Assert linear assumption for subsequent proof steps. Linear means "can be used once".
`*h(a)`	Assert intuitionistic assumption for subsequent proof steps. Intuitionistic means "can be used any number of times".
`-h(X)`	Expectation: consume/apply existing int. assumption.
`=+h(a), =*h(X), =-h(X)`	Timeless versions of the above, meaning that order of assertion of assumptions and their application or consumption can be arbitrary.

A sequential expectation cannot be met by timeless assumption and vice versa, even when they have the same name. In [35], a query cannot succeed with a state which contains an unsatisfied expectation; for simplicity (and to comply with our implementation), this is not enforced in HYPROLOG but can be tested explicitly using a primitive called `expections_satisfied`. Assumption grammars have been used for natural language problems such as free word order, anaphora, coordination, and for knowledge based systems and internet applications.

5.2 Sketch of an example Hyprolog program

A grammar for a subset of English is available at the Hyprolog website [9], click "Sample Hyprolog programs" and then "shootingLuckyLukeAdvanced".[11] It includes pronoun resolution, in which we only allow

[11] There are some mistakes in the version at the Hyprolog website. A corrected one is available at http://www.ruc.dk/~henning/LP-for-Linguists.

backward references, so that "he" can only refer to a male character which has already been mentioned, and "they" can only refer to a group of at least two people already mentioned. The sentence in question refers to a world where people are shooting at each other, and those who have been shot, cannot shoot after that event. Each event is time-stamped according to the sentence in which it appears.[12] Here is an example of a query and its answer; as we see there is only one possible answer.

```
| ?- phrase(discourse, [luckyLuke,shoots,jackDalton,
                        calamityJane,shoots,averellDalton,
                        they,shoot,joeDalton]).
event(2,shooting,[calamityJane,luckyLuke],joeDalton),
event(1,shooting,calamityJane,averellDalton),
event(0,shooting,luckyLuke,jackDalton),
dead(2,joeDalton),
dead(1,averellDalton),
dead(0,jackDalton),
alive(2,luckyLuke),
alive(2,calamityJane),
alive(1,calamityJane),
alive(0,luckyLuke),
'*acting'(masc,joeDalton),
'*acting'(masc,averellDalton),
'*acting'(fem,calamityJane),
'*acting'(masc,jackDalton),
'*acting'(masc,luckyLuke) ? ;
no
| ?-
```

For example, you can see that "they" in the last sentence refers to Calamity Jane and Lucky Luke as they are the only persons mentioned still alive at the time for the shooting. Assumptions such as `'*acting'(masc,jackDalton)` are used for pronoun resolution that also involves the semantic reasoning that only live people can shoot.

As a final example, we illustrate how we can add a context to a discourse, which corresponds to the everyday situation that some amount of common knowledge is assumed, when a conversation begins.

[12] This time-stamping may not be so elegant; I believe it should be possible to get rid of it by using assumptions in the right way.

We can define a context in terms of a Prolog rule, which calls certain constraints and assumptions; here we show only the assumptions in the initial context.

```
duckville:-
*acting(masc,huey),*acting(masc,dewey),
*acting(masc,louie),   *acting(masc,donald),
*acting(fem,daisy).
```

We can use it as follows, where it is applied to pronoun resolution.

```
| ?- duckville, phrase(discourse, [she,shoots,donald]).
event(0,shooting,daisy,donald),
dead(0,donald),
alive(0,daisy),
... ?
```

Note that this principle for setting up an initial context can also be used without the Hyprolog system, so that you can extend the examples shown in the previous sections (avoiding assumptions, of course, which is specific to Hyprolog).

5.3 More reading

The Hyprolog system is the result of my collaboration with Veronica Dahl [13,14], and I have made an implementation which is available at my website, http://www.ruc.dk/~henning/hyprolog/ [9]; there are many examples here that may be useful to inspect. The assumptions in Hyprolog are inspired by Veronica's earlier work [19,34].

6 A few Prolog and CHR systems

There are several good Prolog systems around, some of which may be downloaded for free, but not all include CHR (and occasionally not even DCG).

All examples shown above run in both SICStus Prolog, www.sics.se/sicstus, and SWI Prolog www.swi-prolog.org. SWI is free, but SICStus costs money, although it is available in a test version for 30 days; check if your institution has a site license for SICStus.

You may find a list of all Prologs that support CHR at http://www.cs.kuleuven.be/~dtai/projects/CHR/.

References

1. Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pages 141–152. Physica-Verlag (Springer), 2000.
2. Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!*, volume 7 of *Texts in Computing*. College Publications, 2006. Find also an online version at http://www.learnprolognow.org/.
3. Ivan Bratko. *Prolog (3rd ed.): programming for artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
4. Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. *Commun. ACM*, 38(11):65–70, 1995.
5. CHR web. The programming language CHR, Constraint Handling Rules; official web pages. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`
6. Henning Christiansen. Abductive language interpretation as bottom-up deduction. In Shuly Wintner, editor, *Natural Language Understanding and Logic Programming*, volume 92 of *Datalogiske Skrifter*, pages 33–47, Roskilde, Denmark, July 28 2002.
7. Henning Christiansen. Logical grammars based on constraint handling rules. In Peter J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, page 481. Springer, 2002.
8. Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 5(4-5):467–501, 2005.
9. Henning Christiansen. HYPROLOG: a logic programming language with abduction and assumptions, 2005. Website with source code, User's Guide and examples, http://www.ruc.dk/∼henning/hyprolog/.
10. Henning Christiansen. Logic programming as a framework for knowledge representation and artificial intelligence. Teaching note for the course "Artificial Intelligence and Intelligent Systems", Roskilde University, Denmark, http://www.ruc.dk/~henning/KIIS07/CourseMaterial/CourseNote.pdf, 2006.
11. Henning Christiansen. Implementing probabilistic abductive logic programming with constraint handling rules. In Schrijvers and Frühwirth [32], pages 85–118.
12. Henning Christiansen. Executable specifications for hypothesis-based reasoning with prolog and constraint handling rules. *J. Applied Logic*, 7(3):341–362, 2009.
13. Henning Christiansen and Veronica Dahl. Assumptions and abduction in Prolog. In Elvira Albert, Michael Hanus, Petra Hofstedt, and Peter Van Roy, editors, *3rd International Workshop on Multiparadigm Constraint Programming Languages, MultiCPL'04; At the 20th International Conference on Logic Programming, ICLP'04 Saint-Malo, France, 6-10 September, 2004*, pages 87–101, 2004.

14. Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2005.
15. Henning Christiansen and Verónica Dahl. Meaning in Context. In Anind Dey, Boicho Kokinov, David Leake, and Roy Turner, editors, *Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05)*, volume 3554 of *Lecture Notes in Artificial Intelligence*, pages 97–111, 2005.
16. Henning Christiansen, Christian Theil Have, and Knut Tveitane. From use cases to UML class diagrams using logic grammars and constraints. In G. Angelova, K. Bontcheva, R. Mitkov, N. Nicolov, and N. Nikolov, editors, *RANLP 2007, International Conference: Recent Advances in Natural Language Processing: Proceedings*, pages 128–132. Shoumen, Bulgaria: INCOMA Ltd, 2007.
17. Henning Christiansen, Christian Theil Have, and Knut Tveitane. Reasoning about use cases using logic grammars and constraints. In Henning Christiansen and Jørgen Villadsen, editors, *Proceedings of the 4th International Workshop on Constraints and Language Processing, CSLP 2007*, volume 113 of *Computer Science Research Report*, pages 40–52. Roskilde University, 2007.
18. Alain Colmerauer. Metamorphosis grammars. In Leonard Bolc, editor, *Natural Language Communication with Computers*, volume 63 of *Lecture Notes in Computer Science*, pages 133–189. Springer, 1978. (Translation of an earlier report from 1975 in French: Les grammaires de métamorphose).
19. Verónica Dahl, Paul Tarau, and Renwei Li. Assumption grammars for processing natural language. In *ICLP*, pages 256–270, 1997.
20. Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer, 2002.
21. Peter A. Flach and Antonis C. Kakas, editors. *Abduction and Induction: Essays on their relation and integration*. Kluwer Academic Publishers, April 2000.
22. Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
23. Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, August 2009.
24. Thom W. Frühwirth. User-defined constraint handling. In David Scott Warren, editor, *ICLP*, pages 837–838. MIT Press, 1993.
25. Thom W. Frühwirth. Constraint handling rules: the story so far. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 13–14. ACM, 2006.
26. Jerry R. Hobbs, Mark E. Stickel, Douglas E. Appelt, and Paul A. Martin. Interpretation as abduction. *Artificial Intelligence*, 63(1-2):69–142, 1993.

27. A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pages 235–324, 1998.
28. Antonis C. Kakas and Paolo Mancarella. Database updates through abduction. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *VLDB*, pages 650–661. Morgan Kaufmann, 1990.
29. Robert A. Kowalski. *Logic for problem solving*. Elsevier North Holland, 1979.
30. Shan-Hwei Nienhuys-Cheng and Ronald de Wolf, editors. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science*. Springer, 1997.
31. Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, 1980.
32. Tom Schrijvers and Thom W. Frühwirth, editors. *Constraint Handling Rules, Current Research Topics*, volume 5388 of *Lecture Notes in Computer Science*. Springer, 2008.
33. Swedish Institute of Computer Science. SICStus Prolog user's manual, Version 3.12. Most recent version available at `http://www.sics.se/isl`, 2004.
34. Paul Tarau, Verónica Dahl, and Andrew Fall. Backtrackable state with linear assumptions, continuations and hidden accumulator grammars. In John W. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium*, page 642. MIT Press, 1995.
35. Paul Tarau, Verónica Dahl, and Andrew Fall. Backtrackable state with linear affine implication and assumption grammars. In Joxan Jaffar and Roland H. C. Yap, editors, *ASIAN*, volume 1179 of *Lecture Notes in Computer Science*, pages 53–63. Springer, 1996.
36. David H. D. Warren. An abstract prolog instruction set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.