Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

Assignment #1

To write and query CHR:

- Online using: http://chr.informatik.uni-ulm.de/~webchr/
- Install SWI-Prolog or SICStus Prolog

Exercise 1 (SWI-Prolog)**.** Make yourself familiar with *SWI-Prolog* (which is installed in the labs).

- Read the SWI-Prolog manual on how to use the CHR library:
  http://www.swi-prolog.org/man/chr.html
- Before using CHR rules, the CHR library must be included by:
  `:- use_module(library(chr)).`
- User-defined constraints must be declared with their name and arity, as:
  `:- chr_constraint name/arity.`
- Write a "Hello world!" program in CHR(SWI Prolog):

```
:- use_module(library(chr)).
:- chr_constraint hello/0.

hello <=> write('Hello world!').
```

  Run the program with the query: `hello.`
- Use `chr_trace` to switch on the tracer and view the interactions of the constraint store.

Exercise 2. Implement the following programs which consist of *one* rule each. (Remember to insert an adequate program header!)

- `p1 @ p <=> q.`
- `p2 @ p ==> q.`
- `p3 @ p,q <=> true.`
- `p4 @ p \ q <=> true.`

For each program, try the following queries:

(1) `p`
(2) `q`
(3) `p,p`
(4) `q,q`

Explain the different answers of the system.

Exercise 3. Implement the following programs which consist of *one* rule each.

- `p1 @ p(a) <=> true | true.`
- `p2 @ p(X) <=> X=a | true.`
- `p3 @ p(X) <=> true | X=a.`
- `p4 @ p(X) <=> true , X = a | true.`
- `p5 @ p(X) <=> X = a , X = b | true.`

For each program, pose the following queries:

(1) `p(a)`
(2) `p(b)`
(3) `p(X)`

Explain the different answers of the system.

Exercise 4. Implement the following programs which consist of *one* rule each.

- `p1 @ p(X,Y), q(Z,Y) <=> q(X,Y).`
- `p2 @ q(Z,Y), p(X,Y) <=> q(X,Y).`
- `p3 @ p(X,Y), q(Z,Y) ==> q(X,Y).`
- `p4 @ p(X,Z)\ q(Z,Y) <=> q(X,Y).`

For each program, pose the following queries:

(1) `p(A,B), q(B,C)`
(2) `p(a,b), q(b,c)`
(3) `p(a,b), q(b,c), p(d,a)`
(4) `p(y,c), q(c,a), q(c,a)`

Explain the different answers of the system.

Exercise 5 (Summation)**.** Compute the summation of a multiset of numbers n_i, where numbers are given as query `num(`n_1`), num(`n_2`),..., num(`n_k`)`.
Implement the corresponding CHR rule.

Exercise 6 (Minimum)**.** Compute the minimum of a multiset of numbers n_i, where numbers are given as query `min(`n_1`), min(`n_2`),..., min(`n_k`)`.

Implement the following variations of the problem and test them with appropriate queries:

(1) A program with a single simpagation rule.
(2) Compute minimum from values which should not be removed and trigger computation at a certain point.
(3) Compute several minima from different sources, represented by a constraint: `min(Id,N)`.
(4) Return a computed minimum via a `ismin/1` constraint.

Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

Assignment #2

Exercise 1 (Counting Items)**.** You are given as input a sequence of items in `i/1` constraints, for example `i(bread), i(cheese), i(water), i(bread)`. Your task is to count the items. The output is as `n/2` constraints, for example `n(bread,2), n(cheese,1), n(water,1)`.

Exercise 2 (Color Mixing)**.** Represent colors as propositional CHR constraints `red, blue,...`. Write simplification rules that describe the result of mixing two primary colors, e.g.
`red, yellow <=> orange.`
Observe what happens if you have all three primary colors, in different orders, in the query. How do you ensure that the answer is always the same, say `brown`?

Exercise 3 (Hobo Cigarettes)**.** A certain hobo can make one cigarette out of four cigarette butts (the butt is what is left after smoking a cigarette). If he finds some cigarettes and some cigarette butts, how many cigarettes can he smoke in total?
The input is any number of constraints of this form: `cigarette/0`, `butt/0`, and `pack/1`. Each cigarette represents one cigarette, and each butt represents one cigarette butt. The constraint `pack(N)` represents a pack containing `N` cigarettes. The output is one constraint of the form `smoked(T)`, where T is the total number of cigarettes the hobo has smoked. There can be no cigarettes left (the hobo smokes every cigarette he finds or makes), but there can be (will be) cigarette butts left (always less than four). Examples:

```
?- pack(4).
butt
smoked(5)
```

```
?- pack(25).
butt
smoked(33)
```

```
?- butt, pack(3).
butt
smoked(4)
```

Exercise 4 (Geometry)**.** In an example from geometry, assume that lines are given by variables (or constants) and that CHR constraints express the relationships between two lines, `parallel` and `orthogonal`. Write rules that derive further such relationships from the given relationships, e.g.
`parallel(L1,L2), parallel(L2,L3) ==> parallel(L1,L3).`
Ensure termination.

Exercise 5 (Hamming's problem)**.**
Consider the classical *Hamming's problem*, which is to compute an ordered ascending chain of all numbers whose only prime factors are $2, 3$, or 5. The chain starts with the numbers:

`1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,...`

Define a non-terminating process `hamming(N)` that will produce the numbers as elements of the infinite chain starting with 1. *Hints:*

- pretend that the sequence is already known
- no base cases in recursion as sequences are infinite
- concurrent-process network, processes can be executed in parallel
- to view the stream, you will need an `observe/1` rule that writes elements of the stream as they are produced

Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

Assignment #3

Please complete this assignment and send it by e-mail to amira.zaki@uni-ulm.de, latest by 30.05.12 at 12:00

Exercise 1 (Flattening)**.** Define `eq` to be a binary CHR constraint in infix notation denoting equality, using `op/3`. Write a CHR rule that implements the flattening function that transforms an atomic equality constraint `X eq T`, where `X` is a variable and `T` is a term, into a conjunction of equations as follows, where $X_1, \ldots, X_n$ are new variables.:

$$\texttt{flatten}(X\ \texttt{eq}\ T) := \begin{cases} X\ \texttt{eq}\ T & \text{if } T \text{ is a variable} \\ X\ \texttt{eq}\ f(X_1, \ldots, X_n)\ \wedge\ (\bigwedge_{i=1}^{n}[X_i\ \texttt{eq}\ T_i]) & \text{if } T\ =\ f(T_1, \ldots, T_n) \end{cases}$$

Hints:

- Change terms to list of functor and arguments by: `f(X1,...,XN)=..[f,X1,...,XN]`
- Use the `append/2` predicate that concatenates a list of lists

Implement a second version of the flattening rule, implemented by having a `flatten/2` which flattens a term passed as the first argument into a list (as the second argument).

Exercise 2 (Translating TRS to CHR)**.** It is required to translate TRS rules into CHR simplification rules. Implement a CHR constraint `trs_translate/1` that transforms a TRS rule:

$$S\ \texttt{-->}\ T$$

to a CHR simplification rule:

$$\texttt{flatten}(X\ \texttt{eq}\ S)\ \texttt{<=>}\ \texttt{flatten}(X\ \texttt{eq}\ T)$$

where X is a new variable. The rule should write the output CHR simplification rules to the console. Test your translator with appropriate examples.

Modify the rule such that it writes the CHR rules to a text file, through a new constraint `trs_translate_tofile/1`. Then enhance your program with an additional constraint capable of reading an input text file containing TRS rules and producing a translated CHR program written in an output text file.

Hints:

- You will need to define the binary operator `(-->)`
- `numbervars/2` can be used to pretty print a term by unifying variables ($\_123, \_456, \ldots$) to more readable variable names ($A, B, \ldots$)
- Use `open/3`, `read/2`, `write/2` and `at_end_of_stream/1` to read and write to file streams

Exercise 3 (Translate to CHR)**.** Two rewrite rules that define the addition of natural numbers in successor notation, are:

```
0+Y --> Y.
s(X) + Y --> s(X+Y).
```

Translate the rules into CHR using by applying the flattening function manually. Then use your translator to show its output result. Include in your program 6 appropriate test examples to show the correctness of your work; these examples can be present as comments.

Exercise 4 (Propositional Logic)**.** Given the following TRS for conjunction in propositional logic, where `X`, `Y` and `Z` are propositional variables and the function `and(X,Y)` stands for `X` $\wedge$ `Y`:

```
and(0,Y) --> 0.
and(X,0) --> 0.
and(1,Y) --> Y.
and(X,1) --> X.
and(X,X) --> X.
```

Write down similar TRS rules for negation, `neg`, and disjunction, `or`, in propositional logic. Run the translator and produce the equivalent CHR rules for conjunction, disjunction and negation. Include in your program 6 appropriate test examples to show the correctness of your work; these examples can be present as comments.

Exercise 5 (Functional Dependency)**.** Augment your program with a functional dependency simpagation rule that implements structure sharing to ensure completeness. This CHR rule must be come first within the program.

```
fd @ X eq T \ Y eq T <=> X=Y.
```

The `fd` rule removes equations, thus some rules may not be applicable anymore. For example, for the TRS rule:

```
and(X,X) --> X
```

which translates in the CHR rule:

```
T eq and(T1,T2), T1 eq X, T2 eq X <=> T eq X.
```

The rule expects two copies of the equations `T1 eq X` and `T2 eq Y`. Thus variants of the existing CHR rules must be created, where head constraints have been unified such that the rules apply after the `fd` rule has fired. For the previous example, this results in the additional rule:

```
T eq and(T1,T1), T1 eq X <=> T eq X.
```

Manually write down all other additionally required rules for the other conjunction and disjunction rules.

Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

Assignment #3

Exercise 1 (Adding Natural Numbers)**.** Natural numbers can be denoted in successor notation, e.g. the number `3` is denoted as `s(s(s(0)))`. Write a program that adds natural numbers. Use a constraint `sum/1` for the resulting sum and `add/1` for each number to add. You may assume that one constraint `sum(0)` is present in the query.
E.g. the input `add(s(0)),add(s(s(0))), sum(0)` should produce the output `sum(s(s(s(0))))`.

Exercise 2 (Exchange Sort)**.** An array can be represented as a multiset of pairs of the form `a(Index,Value)`. Implement the exchange sort algorithm that sorts the array of numbers by exchanging values at positions that are in the wrong order. Test it with an appropriate query. What happens if the rule is run backwards, i.e. head and body of the rule are exchanged and guard changed accordingly?

Exercise 3 (Newton's Method)**.** Implement Newton's method for approximating square roots that are related by the following formula:

$$G_{i+1} = (G_i + X/G_i)/2$$

Note that the rule application should stop if the approximation is sufficiently exact. Test the implementation with several examples.

Modify the program such that the approximation cycle is executed exactly 20 times, regardless of the accuracy of the approximation.

Exercise 4 (Reciprocal)**.** Implement the approximation of the reciprocal using the formula:

$$G_{i+1} = 2G_i - XG_i^2$$

Exercise 5 (Days of the Year)**.** Compute the number of days in a given year. The calculation is based on the modulo operation for the year, and additional rules to find out if the given year is a leap year. For example, consider the years 1991, 2000 and 2004.

Exercise 6 (Shortest Paths)**.** The following program takes a directed graph, where the edges are represented as `e/2` constraints (`e(A,B)` means that there is a (directed) edge from `A` to `B`), and computes the reachability relation `p/2` (where `p(A,B)` means that there is some path from `A` to `B`).

```
e(X,Y) ==> p(X,Y).
p(X,Y) \ p(X,Y) <=> true.
e(X,Y), p(Y,Z) ==> p(X,Z).
```

Modify the above program such that it computes the distance relation `d/3`, where `d(A,B,N)` means that the shortest path to go from `A` to `B` uses `N` edges.

Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

Assignment #4

Functional Programming (FP)

Exercise 1 (Translating FP to CHR)**.** It is required to translate FP rewrite rules into CHR simplification rules. Implement a CHR constraint `fp_translate/1` that transforms an FP rule:

$$S \texttt{ --> } (G) \texttt{ ! } T$$

to a CHR simplification rule:

$$X \texttt{ eq } S \texttt{ <=> } G \texttt{ | flatten(}X \texttt{ eq } T\texttt{)}$$

where X is a new variable. The rule should write the output CHR simplification rules to the console. Test your translator with appropriate examples.

Modify the rule such that it writes the CHR rules to a text file, through a new constraint `fp_translate_tofile/1`. Then enhance your program with an additional constraint capable of reading an input text file containing FP rules and producing a translated CHR program written in an output text file.

Hint: You will need to define the binary operators `(-->)` and `(!)`. Also consider that some FP rules can be written as S `-->` T, which should be translated with a `true` guard.

Exercise 2 (Executing FP in CHR)**.** The CHR program produced by the translator implemented above, requires additional rules for treating data and auxiliary functions that are mapped to built-in constraints.
Thus additionally add to your translator, the production of the following two rules:

```
X eq T <=> datum(T) | X=T.
X eq T <=> builtin(T) | execute(T,X).
```

The implementation of the predicates `datum` and `execute` depend on the set of problems to be modeled. For our needs it will be sufficient to define them as follows; `datum(T)` is a predicate that holds if `T` is a number, and `builtin(T)` is a predicate that holds if `T` is a built-in operation (i.e. + or −) on two operands that are numbers and `execute` evaluates to the Prolog `is` operator, i.e. `execute(T,X)` is equivalent to applying the function `T` with `X` as its result, for example `X eq A+B` leads to `X = A+B`. Implement these predicates as defined.

Modify your translator such that it augments the additional rules to the translated FP rules written in the beginning of the output file. It should also add the required CHR header lines such that the output code is runnable by CHR.

Exercise 3 (Translate to CHR)**.** Two FP rules that define the addition of natural numbers in successor notation, are:

```
0+Y --> Y.
s(X) + Y --> s(X+Y).
```

Translate the rules into CHR using by applying the flattening function manually. Then use your translator using the input FP rules to show its output CHR program. Try the modified translator and include in your program 6 appropriate test examples to show the correctness of your work; these examples can be present as comments.

Exercise 4 (Fibonacci Numbers)**.** Recall that the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two. Write the FP rules that calculate the Fibonacci of a number. Translate the rules into CHR manually. Then use your translator using the input FP rules to show its output CHR program. Try your modified translator to produce the CHR code for the translation of the Fibonacci code. Then try to run the Fibonacci code and test is for obtaining several of the Fibonacci numbers. Include in your submission the new output file produced by your complete translator.

The next exercises are to be submitted by e-mail to: amira.zaki@uni-ulm.de.
The deadline is on 04.06.12 by 10:00.

Exercise 5 (Factorial)**.** The factorial of a non-negative integer n is the product of all positive integers less than or equal to n. Write the FP rules for the factorial problem. Make sure your translator produces the required definitions for the `builtin` and `execute` predicates. Produce the equivalent CHR code by the translator, and run it with appropriate test queries.

Exercise 6 (Translating CHR to FP)**.** Go back through the first examples presented in earlier lecture slides. Which of these CHR examples can be expressed in Functional Programming? Also present their equivalent code in Functional Programming. Which examples cannot be expressed, and why?

Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

Assignment #5

Exercise 1 (Translating GAMMA to CHR)**.** It is required to translate each GAMMA pair into a CHR simplification rule. Implement a CHR constraint `gamma_translate/1` that transforms a GAMMA pair:

$$(c/n, f/n)$$

into a CHR simplification rule:

$$d(x_1), \ldots, d(x_n) \texttt{ <=> } c(x_1, \ldots, x_n) \texttt{ | } f(x_1, \ldots, x_n)$$

where `d/1` is a CHR constraint that wraps the data elements, and `c/1` is a built-in constraint that checks for a certain condition. The function f is manually defined by a simplification rule of the form:

$$f(x_1, \ldots, x_n) \texttt{ <=> } G \texttt{ | } D,\ d(y_1), \ldots, d(y_m).$$

where G is a guard and D are the auxiliary built-ins. The built-in constraint `c/n` is a Prolog test predicate, that can be manually defined explicitly per problem.

Implement a CHR rule for the `gamma_translate` constraint that writes the equivalent CHR simplification rule to the console, assume that the definitions for `f/n` and `c/n` will be done separately.

Exercise 2 (GAMMA Lecture Examples)**.** Test your translator with the following GAMMA examples from the lecture slides:

```
min = (</2, first/2)
gcd = (gcd_check/2, gcd_sub/2)
prime = (div/2, first/2)
```

For each example, define the CHR rule for the function `f/2` and the Prolog predicate for `c/2`, also give the translated CHR rule. Test the produced CHR codes with appropriate queries.

Exercise 3 (Translating CHR to GAMMA - Mergers and Acquisitions)**.** A large company will buy any smaller company. A CHR constraint `company(Name,Value)` can be defined where `Value` is the market value of the company. A rule that describes the merge-acquisition cycle that is observed in the real world is given as:

```
company(Name1, Value1), company(Name2, Value2)
         <=> Value1 > Value2 | company(Name1:Name2,Value1+Value2).
```

Translate the Company Mergers CHR program into GAMMA, stating the CHR rule for the function `f` and the Prolog predicate for `c`. Run the translator on the GAMMA pair, and show that the output CHR rules are semantically equivalent to the initial CHR program. Test the produced CHR codes with appropriate queries.

Exercise 4 (Translating CHR to GAMMA - Walk)**.** Assume we describe a walk (a sequence of steps) by giving directions, `east, west, south, north`. A description of a walk is just a sequence of these CHR constraints. Note that the order in which the steps are made is not important to determine the position reached. With simplification rules, we can model the fact that certain steps (like `east, west`) cancel each other out, and thus we can simplify a given walk to one with a minimal number of steps that reaches the same position.

```
east, west <=> true.
south, north <=> true.
```

Translate the Walk CHR program into GAMMA, stating the CHR rule for the fucntion `f` and the Prolog predicate for `c`. Run the translator on the GAMMA, and show that the output CHR rules are semantically equivalent to the initial CHR program. Test the produced CHR codes with appropriate queries.

The next exercises are to be submitted by e-mail to: amira.zaki@uni-ulm.de.
The deadline is on 12.06.12 by 10:00.

Exercise 5 (Translating CHR to GAMMA - More examples)**.** For each of the following problems, write down the CHR rules to solve them. Then translate the CHR program into GAMMA, stating the CHR rule for the function `f` and the Prolog predicate for `c`. Run the translator on the GAMMA pair, and show that the output CHR rules are semantically equivalent to the initial CHR program. Test the produced CHR codes with appropriate queries.

(1) **Exclusive OR** - A multi-set of `xor` constraints denoting the output can be used to compute the output as a single remaining `xor` constraint where truth values `true` and `false` are represented by the numbers `1` and `0` respectively.

(2) **Exchange Sort** - Sort an array by exchanging values at positions that are in the wrong order, given an unsorted array as a sequence of constraints of the form `a(Index,Value)`.

(3) **Destructive Assignment** - In a declarative programming language, bound variables cannot be updated or overwritten. However, in CHR it is possible to simulate the destructive (multiple) assignment of procedural and imperative programming languages by using recursion in CHR. The original constraint with the old value is removed and a constraint of the same type with the new value is added. For example, we can store variable name-value pairs in the CHR constraint `cell/2` and use the CHR constraint `assign/2` to assign to a variable a new value.

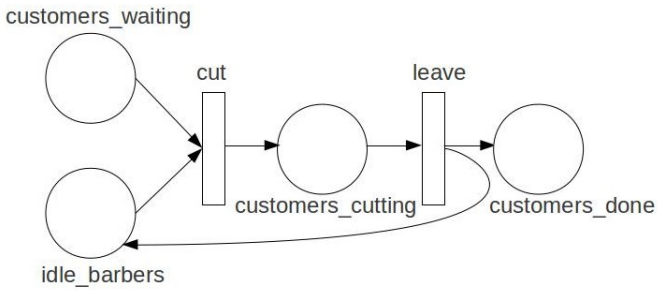
(4) **Merge sort** - To represent a directed edge (arc) from node A to node B, we use a binary CHR constraint written in infix notation, `A -> B`. We use a chain of such arcs to represent a sequence of values that are stored in the nodes, e.g. the sequence `0,2,5` is encoded as `0 -> 2, 2 -> 5`. A one-rule CHR program performs an ordered merge of two chains by zipping them together, provided they start with the same (smallest) node.

Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

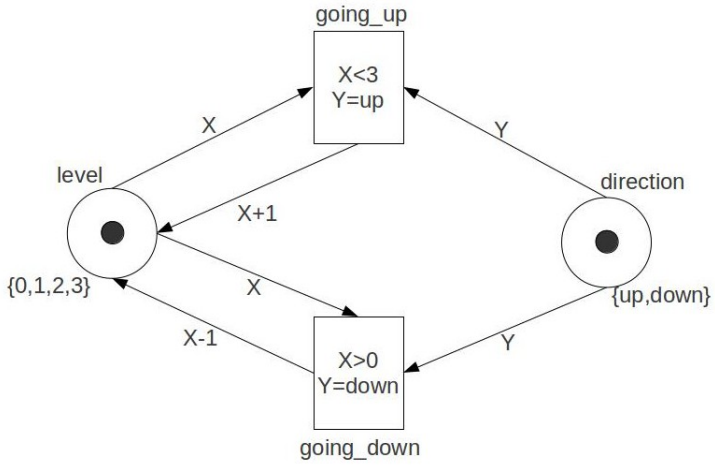
Assignment #6

Exercise 1 (Petri Nets - Barber Shop)**.** A typical scenario at a Barber shop is as follows: Customers enter a Barber shop and wait till a barber is idle and ready to serve them. Then the barber cuts the hair of a customer. When the hair cut is done, the customer leaves and the barber becomes idle once again. This can be represented using the Petri net given below:



Translate the Barber shop Petri net into CHR by adding the constraints `customers_waiting/0`, `idle_barbers/0`, `customers_cutting/0`, and `customers_done/0` for each of the places. Add an `observer/0` constraint to print the interesting states of the problem. A typical test query would be `?-observer,customers_waiting,customers_waiting,idle_barbers.`

Exercise 2 (Colored Petri Nets - Elevator)**.** An elevator operates between 4 levels, a petri-net is designed such that a level token represents the level of the elevator and the direction tokens represent the required movements of the elevator. The elevator can go upwards, increasing its level only if it is not on the maximum floor. Similarly the elevator can go down, decreasing its level only if it is not on the ground floor. The system can be represented using the following colored Petri net:



Translate the elevator Petri net into CHR by adding the constraints `level/1, direction/1` and the equivalent transition rules. Add an `observer/0` constraint to print the interesting states of the problem. A typical test query would be
`?- observer, level(0), direction(up),direction(up),direction(up),direction(down).`

Exercise 3 (Preprocessing OPS5)**.** The OPS5 rule for the iterative Fibonacci sequence generation is given as:

```
(p next-fib (limit ^is <limit>)
   {(fibonacci ^index {<i> <= <limit>}
               ^this-value <v1>
               ^last-value <v2>) <fib>}
   --> (modify <fib> ^index (compute <i> + 1)
                     ^this-value (compute <v1> + <v2>)
                     ^last-value <v1>)
       (write (crlf) Fib <i> is <v1>)
)
```

We define a slight preprocessing step on the input OPS5 rules to make it more readable through Prolog by removing reserved Prolog operators and introducing lists. This involves:

- changing all rounded brackets to be square brackets, i.e. `( )` to `[ ]`
- changing dashes to underscores, i.e. `-` to `_`
- changing all spaces to commas `,`
- removing curly braces, attribute markers and variable delimiters `{ } ^ < >`
- indexing all variables with a capital `V` to translate them into valid Prolog variable names
- removing control statements such as `(halt)` and `(crlf)`
- removing names given to left-hand-side terms and just using the same name with the equivalent right-hand-side ones

To make the next exercise easier, we assume that whenever a modification takes place, all attributes are always given (even if they remain unchanged) and their order is as they were first stated. Applying this preprocessing step on the previous Fibonacci example, it becomes:

```
[p, next_fib,
       [limit, is, Vlimit],
       [fibonacci, index, [Vi =< Vlimit], this_value, Vv1, last_value, Vv2],
       -->,
            [modify, fibonacci, index, [compute, Vi + 1],
                                this_value, [compute, Vv1 + Vv2],
                                last_value, Vv1],
            [write, Fib, Vi, is, Vv1]
]
```

Apply this preprocessing technique on the `Greatest Common Divisor` example of the lecture.

Exercise 4 (Translating OPS5 Production Rules to CHR)**.** An OPS5 production rule:

```
(p N LHS --> RHS)
```

which after applying the preprocessing step becomes:

```
[p,N,LHS,-->,RHS]
```

translates to the CHR generalized simpagation rule:

```
N @ LHS1 \ LHS2 <=> LHS3 | RHS'
```

where `LHS`: if-clause, `RHS`: then-clause, `LHS1`: patterns of `LHS` for facts not modified in `RHS`, `LHS2`: patterns of `LHS` for facts modified in `RHS`, `LHS3`: conditions of `LHS`, and `RHS'`: `RHS` without removal (for `LHS2` facts).

Write an `ops_translate/1` constraint that takes a list containing the preprocessed OPS5 rule, decomposes it into CHR components and writes on the console the equivalent CHR code. Define `rulename/1, constraint/3, guard/1, writes/1` constraints to identify the various parts of the OPS5 rule. The `constraint` consists of its name, arguments and status (kept, removed or added). After decomposing the rule, you can use a sequence of flow-controlled writing constraints to display the various parts of the translated CHR rule in the correct order and format.

Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

Assignment #7

Incremental Conflict Resolution in CHR

Exercise 1 (*Step 1: Translation)*. A generalized CHR simpagation rule (with a property P) which is given as follows:

```
Head1 \ Head2 <=> Guard | Body : P.
```

can be represented using a `chr_rule/5` constraint in the form:

```
chr_rule(Head1, Head2, Guard, Body, P)
```

where `Head1, Head2, Body` are a list of CHR constraints, if they are do not exist (i.e. for a propagation or simplification rule) then they are given as `[true]` and empty guards as `true`.

In the lecture, a translation scheme was discussed which translates the rule into two other CHR rules by introducing a `conflictset/1` constraint to gather rule bodies and then execute the chosen rule from the conflict set. However, the rule notation presented must be extended, such that the rule terms have an arity of 4, i.e. `rule(P,Head1,Head2,Body)`. The output rules can be distinguished from the input ones by representing them using a `chr_rule/4` constraint (i.e. same as `chr_rule/5` but without the 5th argument). Write a translation rule that performs the technique discussed, i.e. changes each `chr_rule/5` constraint into two `chr_rule/4` constraints.

Use and extend the writing rules from Assignment 6 to format the `chr_rule/4` rules into their common CHR form and display them on the console. *(To save time do not perform this task, instead just use the dummy version sent to you by email; for sure its optimization is out of the scope of this exercise. Just invoke it by adding a* `display/0` *constraint to your query).*

Exercise 2 (*Step 2: Additional Conflict Resolution Rules)*. For any set of translated simpagation rules the following additional rules are required to resolve the conflict:

```
collect @ conflictset(L1), conflictset(L2) <=> append(L2,L1,L3), conflictset(L3).
choose  @ fire, conflictset(L) <=> choose(L,R,L1), conflictset(L1), apply(R), fire.
```

Consider the case when the conflict set does not contain any more rules, how would the previous two rules be modified? Then consider the case when the `choose/3` cannot find any applicable rule from the list, how would this change the required additional rules?

Exercise 3 (*Step 3: Additional Choice Rules)*. For the rule choice, the `choose/3` constraint selects a particular rule from the conflict set of rules depending on the property P stated in the initial translated CHR rule.

(1) $P = random$: randomly selects a rule from the conflict set
(2) $P = bfs$: selects a rule from the set to ensure the breadth first traversal of rules
(3) $P = N, number(N)$: selects the rule with the highest priority (P)
(4) $neg(C, G)$: negation as absence; the rule is applied if there are no CHR constraints `C` for which the guard `G` holds

Write three rules for cases 1 to 3, simplifying the `choose/3` constraint such that the first argument contains the list of rule terms (whose first term specifies the choice criterion), the second argument is binded to the selected rule, and the third argument is binded to the remaining list. *(Please note that case 4 will be covered next week).*

Exercise 4 (*Example: Random Dice)*. A dice is thrown and the result can be a 1, 2, 3, 4, 5 or 6. The result is required to be random, thus the random execution of the rules can be encoded using the following set of CHR rules:

```
dice <=> write(1) : random.
dice <=> write(2) : random.
dice <=> write(3) : random.
dice <=> write(4) : random.
dice <=> write(5) : random.
dice <=> write(6) : random.
```

A typical test query and its results would be:

```
?- dice,fire.
   5

?- dice,fire.
   2
```

Perform step 1, by encoding the rules into `chr_rule/5` constraints. Then run the translator from step 1 to obtain the 12 output CHR rules. Create a new file, add the output 12 rules in addition to the rules from steps 2 and 3. Test your code with appropriate queries, and show the multiple random results.

Exercise 5 (*Example: Dijkstra)*. Dijkstra's shortest path algorithm can be expressed by giving priority to the application of the CHR rules. A lower weight is given to shorter paths, while constructing a longer path has more weight and hence its rule would have lower priority. This can be encoded using the following rules:

```
d2 @ dist(X,N) \ dist(X,M) <=> N<M | true : 1.
dn @ dist(X,N), edge(X,Y,M) ==> P is N+2 | Z is N+M, dist(Y,Z) : P.
```

A typical test query would be:

```
?- edge(a,b,10),edge(a,c,2),edge(b,c,1),
   edge(b,a,10),edge(c,a,2),edge(c,b,1),dist(a,0),fire.
```

Perform step 1, by encoding the rules into `chr_rule/5` constraints. Then run the translator from step 1 to obtain the output CHR rules, augment them to the rules from steps 2 and 3. Test your code with appropriate queries and show the results.

Exercise 6 (*Example: Multiple Count-ups)*. The following CHR program which when given a `count/1` constraint, can either display it on the console or calculate the next count. It is required to perform a breadth-first-traversal of the rules. The CHR code is given as following:

```
count(X) ==> write(X) : bfs.
count(X) <=> Y is X+1, count(Y) : bfs.
```

A typical test query and its results would be (but with the output numbers on separate lines):

```
?- count(0),count(100),fire.
   0  100  1  101  2  102  3  103  4  104  5  105  6  106 ...
```

Perform step 1, by encoding the rules into `chr_rule/5` constraints. Then run the translator from step 1 to obtain the output CHR rules, augment them to the rules from steps 2 and 3. Test your code with appropriate queries, and show the multiple random results.

Rule-based Programming

Prof. Dr. Thom Frühwirth
Amira Zaki

Assignment #8

Negation as Absence

Exercise 1 (*Case 4 of Step 3: Negation Choice Rule)***.** As a continuation of Exercise 3 in Assignment 7, it remains to handle negation as absence. In CHR, a rule which checks for the absence of particular constraints is expressed as follows:

```
Heads1 \ Heads2 <=> Guard | Body : neg(NH,NG).
```

Write a rule for the 4th case such that the `choose/3` constraint when given an input list where the rule has `P = neg(NH,NG)` then: if the constraints `NH` are present and `NG` holds, then it should remove the rule with this negation and continue resolving the conflict, otherwise the rule with negation is chosen as the rule to apply.

Exercise 2 (*Example: Martial Status)***.** A person is single or married, where single is to be the default. This can be expressed by a negated CHR rule as follows:

```
person(X) ==> single(X) : neg(married(X),true).
```

A typical test query and its results would be:

```
?- married(a), person(b).
married(a)    person(b)    single(b)
```

As done in Assignment 7, encode the rule into an equivalent `chr_rule/5` constraint, then run the translator to obtain the output CHR rules. Test your code with appropriate queries, and show the multiple results.

Exercise 3 (Example: Minimum as Negation)**.** A person is single or married, where single is to be the default. This can be expressed by a negated CHR rule as follows:

```
num(X) ==> min(X) : neg(num(Y), Y<X).
```

A typical test query and its results would be:

```
?- num(2), num(1), num(10).
num(10)    num(1)    num(2)    min(1)
```

As done in Assignment 7, encode the rule into an equivalent `chr_rule/5` constraint, then run the translator to obtain the output CHR rules. Test your code with appropriate queries, and show the multiple results.

Exercise 4 (Example: Graphs Closure)**.** When finding the transitive closure of a graph, then a negated CHR program can be given as:

```
e(X,Y) ==> p(X,Y) : neg(p(X,Y),true).
e(X,Y), p(Y,Z) ==> p(X,Z) : neg(p(X,Z),true).
```

A typical test query and its results would be:

```
?- e(a,b), e(b,c), e(c,d).
e(c,d)  e(b,c)  e(a,b)  p(a,d)  p(b,d)  p(a,c)  p(c,d)  p(b,c)  p(a,b)
```

As done in Assignment 7, encode the rule into an equivalent `chr_rule/5` constraint, then run the translator to obtain the output CHR rules. Test your code with appropriate queries, and show the multiple results.

Ensuring Set-based Semantics

Exercise 5 (*Rule Variants).* Given any CHR simplification rule (ignoring priorities):

```
H, H1, H2 <=> G | B[,H1,H2].
```

it is possible to generate new rule variants by systemically unifying head constraints in all possible ways:

```
H, H1 <=> H1 = H2, G | B[,H1].
```

The same unify and merge technique can be applied to simpagation and propagation rules; if any of the heads merged was originally to be kept, then it remains kept in the variant rule even if it was unified with a head that would be removed.

Write a rule that transforms a CHR rule written as `chr_rule/4` constraints into all possible variants, by trying to unify and merge all head constraints. *Hint: Use disjunction in the CHR rule body to enforce possible rule firings to obtain the various unify possibilities.* Use the modified version of the `ex8_writer.pl` to output the generated rules on the console. It works by encapsulating heads into `head/2` constraints, where the first argument is the actual head and the second is 1 for kept and 0 for removed constraints, guards into `guard/1` and bodies into `body/1`. A typical test query would be: `chr_rule([p(a,X), p(Y,b)],[p(Z,W)],true,p(X,Z)).` Test your code with similar such appropriate queries, and show the multiple results.

Exercise 6 (*Examples - Homework by 04.07).* For the following list of CHR programs, encode the rule into an equivalent `chr_rule/4` constraint, then produce all possible rule variants. Examine the produced rules and justify if they are necessary, redundant, non-terminating or incorrect.

(1)
```
minimum @ min(Y) \ min(X)<=> Y=<X | true.
```

(2)
```
gcd @ gcd(N) \ gcd(M) <=> (0<N,N=<M) | X is M-N, gcd(X).
```

(3)
```
xor1 @ xor(X), xor(X) <=> xor(0).
xor2 @ xor(0), xor(1) <=> true.
```

(4)
```
primes @ prime(A) \ prime(B) <=> B mod A=:=0 | true.
```

(5)
```
transitive_closure @ p(A,B), p(B,C) ==> true | p(A,C).
```

(6)
```
summing @ accu(X), accu(Y) <=> Z is X+Y, accu(Z).
```

(7)
```
sort @ X <<< A \ X <<< B       <=> A<B | A <<< B.
merge @ merge(N,A), merge(N,B) <=> A<B | M is N+1, merge(M,A), A <<< B.
```

(8)
```
antisymmetry @ leq(X,Y), leq(Y,X)  <=> X = Y.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y), leq(Y,Z)  ==> leq(X,Z).
```

(9)
```
le @ X le Y, X::A:_, Y::_:D ==> Y::A:D, X::A:D.
eq @ X eq Y, X::A:B, Y::C:D ==> Y::A:B, X::C:D.
ne @ X ne Y, X::A:A, Y::A:A <=> fail.
```

(10)
```
mult_z @ mult(X,Y,Z), X::A:B, Y::C:D ==>
            M1 is A*C, M2 is A*D, M3 is B*C, M4 is B*D,
            Z::min(min(M1,M2),min(M3,M4)):max(max(M1,M2),max(M3,M4)).
```