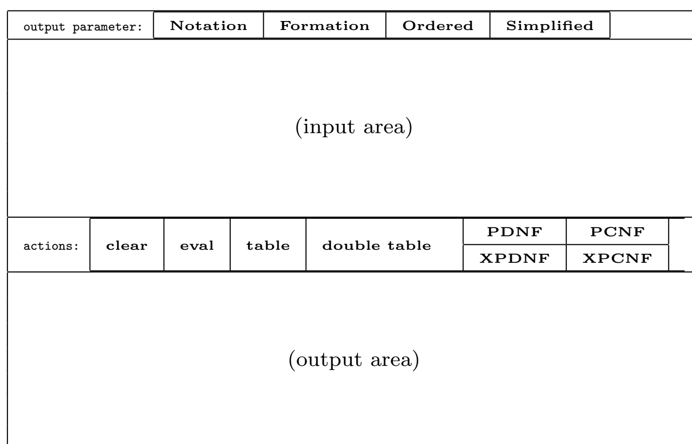
Bucanon introduction: How to do boolean algebra with the bucanon program

www.bucephalus.org

September 5, 2002

The working panel

The **Bucanon panel** is the working area for the user of both the Bucanon applet and the Bucanon application. It has the following structure:



From top to bottom the Bucanon panel has four areas:

- A row of **output parameter** buttons, where the settings for the output can be altered. For now you don't need to bother with their meaning and just leave all the settings to their default values.
- An **input area**. Here is the place to insert your input formulas. In particular these can be boolean formulas, which are defined below.
- A row of **action** buttons. After the input of a formula, one of the actions can be applied to it. The **clear** button deletes every previous input and output. Furthermore, the following actions are of interest and sufficient to (efficiently) solve all problems and questions of boolean algebra / propositional logic:

  eval **table** **PDNF** **PCNF**

  We will explain them below.
- The **output area**, i.e. the place where the results appear.

Syntax of boolean formulas

Boolean formulas are strings which are defined according to the rules given in figure 1. So one example of a boolean formulas is given by

```
[a , -b , [c -> d] , !  , '[a ; 'b ; !]  , c]
```

boolean formula φ	
atom α	non-empty string of letters (A,..,Z,a,...,z), digits (0,1,...,9), and the understroke (_)
boolean junction	
bit value	
zero bit	?
unit bit	!
negation (in stroke notation)	'φ
negation (in arithmetic notation)	-φ
conjunction (in stroke notation)	[,] or [,φ] or [φ_1,...,φ_n] with $n \geq 2$
conjunction (in arithmetic notation)	[*] or [*φ] or [φ_1*...*φ_n] with $n \geq 2$
disjunction (in stroke notation)	[;] or [;φ] or [φ_1;...;φ_n] with $n \geq 2$
disjunction (in arithmetic notation)	[+] or [+φ] or [φ_1+...+φ_n] with $n \geq 2$
subjunction	[φ -> φ]
equijunction	[φ <-> φ]

Figure 1: syntax of boolean formulas

standing for

"a and not b and (if c then d) and true and not(a or not b or true) and c".

Note that:

- There are two alternative notations for negation, conjunction and disjunction:

	stroke notation	arithmetic notation
negation	'x	-x
conjunction	[,] or [,x] or [x_1,...,x_n] with $n \geq 2$	[*] or [*x] or [x_1*...*x_n] with $n \geq 2$
disjunction	[;] or [;x] or [x_1;...;x_n] with $n \geq 2$	[+] or [+x] or [x_1+...+x_n] with $n \geq 2$

  You can use either notation, even in a mixed mode like in `['a , -b , [c + -d] , -a]` for "not a and not b and (c or not d) and not a". But note that you cannot change the notation inside one junction, e.g. `[a , b * c , d]` for "a and b and c and d" is not permitted and generates an error message. The output is always either completely in stroke or completely in arithmetic notation. The default is stroke notation and you can change it with the *notation* ouput parameter button.
- The use of ? for **zero** or **false** and ! for **unit** or **true** in the Bucanon syntax is less common. As a rule to memorize these symbols, you can think of the shape of "?" as "circle and dot", which is "zero bit". Accordingly "!" is "stroke and dot", which is "unit bit".
- The bracket symbols are part of the syntax and cannot be left or added arbitrarily. For example, the input of `[a]` for `a` or `a * b` for `[a * b]` is not correct.

table

The **table** action displays the **bit value** or **truth table** of the given input formula. For example for the formula `[a , c , 'b]`, standing for "a and c and not b", the output will be

a	b	c	[a , c , 'b]
?	?	?	?
!	?	?	?
?	!	?	?
!	!	?	?
?	?	!	?
!	?	!	!
?	!	!	?
!	!	!	?

eval

The **eval** action basically means that the bit values are eliminated in boolean formulas according to the usual rules for the boolean junctors. For example

- `[ !  , ?  ]` for "true and false" becomes `?`, i.e. "false" after evaluation,
- `[ -!  + ?  + ?]` for "not true or false or false" turns into `?`,
- `[ 'a , ?]` for "not a and false" becomes `?`, while
- `[ 'a , !]` evaluates to `'a` and
- `[a -> [b * ?]]` for "if a then (b and false)" first evaluates to `[a -> ?]`and finally becomes `'a`.

You can also check if two boolean formulas φ_1 and φ_2 are sub– or equivalent:

- Input the formula

  `[`φ_1 `=>` φ_2`]`

  and after the evaluation either `!` or `?` will be the result, depending on the fact if φ_1 does entail φ_2 or not.
  For example the input `[ [a,b] => a ]` evaluates to `!`, since it is "true" that "(a and b) entails a".
- Accordingly one can determine the equivalence of φ_1 and φ_2 by evaluating the input

  `[` φ_1 `<=>` φ_2 `]`

  For example `[ '[a,b] <=> ['a;'b] ]` turns into `!` after evaluation, because it is true, due to de Morgans law, that "not(a and b) is equivalent to (not a or not b)".

The equivalence can also be used to determine if a boolean formula φ is valid etc:

- φ is **valid** or a **tautology** if and only if `[`φ `<=> !]` evaluates to `!`.
- φ is **satisfiable** if and only if `[`φ `<=> ?]` evaluates to `?`.
- φ is a **contradiction** if and only if `[`φ `<=> ?]` evaluates to `!`.

Occasionally it may also be useful to list all the atoms occuring in a formula φ. This is done by evaluating the input form

`@`φ

For example,

```
@[d , 'b , a , '['b + !  + [a -> b] + d]]
```

evaluates to

```
[a b d]
```

The result is always an ordered list, according to a predefined linear order between atoms.

These methods for generating tables, evaluating sub– and equivalences and deciding the validity of formulas are the standard straight forward methods of propositional logic. But they are not practical for less trivial cases in general, because their computational costs explode exponentially: It takes up to 2^n recursive steps to decide the validity of a boolean formula with n atoms.

PDNF and PCNF

Much more efficient is the application of the two **boolean canonizers PDNF** and **PCNF** for the generation of the unique **prime disjunctive / conjunctive normal form** of a given input formula.

A **disjunctive normal form** or **DNF** has the following form (we use the default stroke notation in the sequel):

`[ [`$\lambda_{1,1}$`,...,`λ_{1,n_1}`] ; ... ; [`$\lambda_{m,1}$`,...,`λ_{m,n_m}`] ]`

where each $\lambda_{i,j}$ is a **literal**, i.e. either an atom α or a negated atom `'`α. In our definition we additionally demand that each of the literal conjunctions `[`$\lambda_{i,1}$`,...,`λ_{i,n_i}`]` is **normal**, i.e. ordered in the sense that $\alpha_{i,1} < \cdots < \alpha_{i,n_1}$ for the atoms of the given literals.

An example of such a DNF is given by

```
[ ['a,b,d] ; [a,b,'c,d] ; ['b,'d] ; ['a,c,'d] ; ['c,d] ]
```

We say that a normal literal conjunction γ = `[`λ_1`,...,`λ_n`]` is a **prime factor** of a given formula φ, if γ itself is subvalent to φ (i.e. the evaluation of `[`γ `=>` φ`]` is **!**) and each

elimination of literals violates this relation (i.e. $[[\lambda_1,\lambda_{i-1},\lambda_{i+1},\ldots,\lambda_n]$ => $\varphi]$ evaluates to ?, for every $i = 1, \ldots, n$).

A **prime DNF** or **PDNF** now is defined as a DNF $\Delta = [\gamma_1,\ldots,\gamma_m]$, where $\{\gamma_1,\ldots,\gamma_m\}$ is exactly the set of all its prime factors. It can be shown that every formula φ has an equivalent PDNF Δ, and this form Δ is unique (up to the order of its arguments $\gamma_1,\ldots,\gamma_m$). It is generated by the according Bucanon action button after the input of φ.

For example, let φ be the input formula `[ [a,b] -> 'a]`. Its PDNF is given by `[ [,'b] ; [,'a] ]`.

This example PDNF is a proper DNF, of course, but it looks a bit artificial. It is custom in propositional logic not to use unary conjunctions like `[,'b]` and `[,'a]` and rather write them as `'b` and `'a`. We call this conventional form the **simplified** version. The "simplified / not simplified" option is one of the Bucanon output parameter settings and it is set to "simplified: yes" by default. So the simplified version of our example output becomes `['b ; 'a]`, and that is what the program returns.

Simplified basically means that unary conjunctions and disjunctions are eliminated and that nullary junctions like `[,]` and `[;]` are replaced by the bit values `!` and `?`, respectively. Accordingly, the (P)DNF `[;[,]]` is replaced by `!`.

Next to the PDNF canonization there is the **PCNF** canonization that generates the **prime conjunctive normal form** of the given input formula, which has the form

$$[\ [\lambda_{1,1};\ldots;\lambda_{1,n_1}]\ ,\ \ldots\ ,\ [\lambda_{m,1};\ldots;\lambda_{m,n_m}]\ ]$$

in its non–simplified version and in stroke notation.

The two boolean canonizers can be efficiently applied to the standard problems as follows (we assume that the *simplified* output parameter is set to its default value *yes*):

- Let φ_1 and φ_2 be two boolean formulas.
  - $[\varphi_1$ => $\varphi_2]$ is true iff the PDNF of $[\varphi_1$ -> $\varphi_2]$ is ! iff the PCNF of $[\varphi_1$ -> $\varphi_2]$ is !.
  - $[\varphi_1$ <=> $\varphi_2]$ is true iff the PDNF of $[\varphi_1$ <-> $\varphi_2]$ is ! iff the PCNF of $[\varphi_1$ <-> $\varphi_2]$ is !.
- Let φ be a boolean formula.
  - φ is valid iff the PDNF of φ is ! iff the PCNF of φ is !.
  - φ is satisfiable iff the PDNF of φ is different from ! iff the PCNF of φ is different from !.
  - φ is a contradiction iff the PDNF of φ is ? iff the PCNF of φ is ?.