# CHAPTER 10

## Comparison of Clausal Form with Standard Form

Clausal form is simpler than the standard form of logic and bears greater resemblance to other formalisms used for databases and programming. Moreover, the resolution rule resembles conventional rules for information processing and problem-solving more closely than does standard form.

Although any problem can be converted from standard form to clausal form, the standard form is often more economical and more natural than the resulting collection of clauses. The specification of programs, in particular, is an area in which the standard form of logic (or some appropriate extension of Horn clause form) is more suitable than simple clausal form. Moreover, the derivation of programs from specifications can be achieved more naturally by reasoning with the standard form of logic directly. Useful inference systems for the standard form of logic, however, may be obtained by combining inference rules for clausal form with rules for converting from standard form to clausal form.

### Introduction to the standard form of logic

We shall present only the informal semantics of the standard form of logic, by associating expressions of English with expressions of the symbolic language. Such notions as "consistency" for expressions in standard form can be understood informally in terms of their English language counterparts.

The standard form of logic provides explicit symbolism for the propositional connectives "and", "or", "not", "if" and "if and only if" and for the quantifiers "for all" and "there exists". The propositional connectives construct more complex propositions from simpler ones. The symbol

& stands for "and"
V stands for "or"
¬ stands for "not"
-> stands for "if... then... " or "implies"
<-> stands for "if and only if".

A clause

$$A_1,...,A_m \leftarrow B_1,...,B_n$$

not containing variables, is written

$$[B_1\&...\&B_n] \rightarrow [A_1 \vee ... \vee A_m]$$

in standard form. If n=0, the standard form omits the arrow

$$A_1 \vee ... \vee A_m$$

If m=0, the arrow becomes a negation symbol.

$$\neg[B_1\&...\&B_n].$$

In standard form the direction of the implication sign -> is opposite to the one we have been using in clausal form. But like the inequality sign < or > of arithmetic the direction of the implication sign is not significant. Thus the expressions

A -> B and B <- A

are equivalent. But notice that

A -> B and A <- B

are not.

Sentences in standard form can also be constructed by means of the two quantifiers.

The <u>universal quantifier</u>

∀x stands for "for all x".

The <u>existential quantifier</u>

∃x stands for "there exists an x".

<u>Example</u> Some oysters can be crossed in love.

<u>Clausal Form</u> Oyster(⧗) <-
Crossed-in-Love(⧗) <-

<u>Standard form</u> ∃x [Oyster(x) & Crossed-in-Love(x)]

In the clausal formulation, in order to refer to an individual, it is necessary to give it a name. The existential quantifier allows individuals to be referred to without being named. In clausal form sentences are implicitly connected by "and". In standard form the conjunction & can be written explicitly.

<u>Example</u> Every human has a mother.

<u>Standard Form</u> ∀x∃y[Human(x) -> Mother(y,x)]

<u>Clausal Form</u> Mother(mum(x),x) <- Human(x)

In the clausal form it is necessary to use a function symbol to name the individual y which exists as a function of x.

Changing the order of the quantifiers changes the meaning. The sentence

∃y∀x[Human(x) -> Mother(y,x)]

states there is a single individual who is the mother of us all. The clausal form uses a constant symbol to name the individual.

Mother(☺,x) <- Human(x)

For the precise definition of sentence, it is necessary to define the more general notion of formula. Formulae may contain free (unquantified) variables, whereas sentences do not. Thus the formula

∀x∃yLoves(x,y)

is a sentence, but the formula

∀xLoves(x,y)

is not. It contains the bound (quantified) variable x and the free variable y.

*Terms* and *atomic formulae* are defined just as for clausal form.

An expression Z is a *formula* if and only if it is an atomic formula or an expression of the form

[X & Y]
[X V Y]
[X -> Y] or [Y <- X]
[X <-> Y]
¬ X
∀v X or
∃v X

where X and Y are formulae and v is any variable.

Any formula Z is a *subformula* of itself. In the first four cases above, any subformula of X or Y is a *subformula* of Z; and in the last three cases, any subformula of X is a *subformula* of Z.

An occurrence of a variable v in a formula Z is *free* (or unbound) if it belongs to no subformula of Z of the form ∀v X or ∃v X. If an occurrence of v is free in X then it is *bound* in ∀v X and ∃v X by the quantifiers ∀v or ∃v respectively.

A formula is a *sentence* if and only if it contains no free occurrence of a variable.

The definitions above permit sentences such as

∃x [Oyster(x) & ∃x Tasty(x)]

in which the same variable x is bound by different occurrences of a quantifier. Such sentences create complications which are better avoided. Consequently we shall restrict formulae Z to those which satisfy the condition that

> for every variable v which occurs in Z, either all occurrences of v in Z are free in Z or all occurrences of v in Z are bound by the same quantifier occurrence.

Any formula Z which violates the restriction can be transformed into an equivalent one which satisfies it by renaming variables. This can be done by applying the equivalences

| |
|---|
| ∀u X <-> ∀v X' |
| ∃u X <-> ∃v X' |
| where X' is obtained from X by replacing all occurrences of u by v and v does not occur in X. |

to subformulae of Z. Any subformula can be replaced by an equivalent one without affecting the meaning of the formula in which it occurs.

Notice also that the definitions permit quantification ∀v X or ∃v X of a variable v which does not occur in the formula X. Such quantification is vacuous in the sense that the resulting formula is equivalent to the unquantified formula X. Deletion of vacuous quantifiers is justified by the equivalences:

| |
|---|
| ∀v X <-> X |
| ∃v X <-> X |
| where the variable v does not occur in X. |

Several conventions can be employed to improve the readability of formulae by reducing the number of brackets. Outermost brackets can always be omitted, writing A -> B, for example, rather than [A -> B].

The associativity of conjunction justifies omitting brackets when several formulae are conjoined together. Since the formulae

A & [B & C] and
[A & B] & C

are equivalent, it is permissible to ignore brackets altogether, writing

A & B & C.

Similarly, the associativity of disjunction justifies writing

A ∨ B ∨ C
instead of A ∨ [B ∨ C] or
[A ∨ B] ∨ C.

Brackets can be reduced further by establishing precedence rules for the quantifiers and the propositional connectives. We shall follow the conventions that

> The *negation symbol* ¬ and the *quantifiers* ∃, ∀ bind more closely than the other symbols and *conjunction* & and *disjunction* V bind more closely than *implication* -> and *equivalence* <->.

Thus we may safely write

A V B V C <- D & E & F

instead of [[A V [B V C]] <- [[D & E] & F]]

for example.

Readability can be improved further by omitting universal quantifiers at the beginning of sentences, writing, for example,

Grandparent(x,y) <- Parent(x,z) & Parent(z,y)

instead of ∀x∀y∀z [Grandparent(x,y) <- Parent(x,z) & Parent(z,y)]

as in clausal form. Such omission of universal quantifiers can be performed safely only when the context makes it clear that the expression is a sentence rather than a formula containing occurrences of free variables.

## Conversion to clausal form

Any sentence in standard form can be converted to clausal form. The resulting set of clauses is consistent if and only if the sentence in standard form is consistent. Thus conversion to clausal form can be used to demonstrate the inconsistency of a set of sentences in standard form:

> A set of sentences in standard form is inconsistent if and only if the corresponding set of clauses is inconsistent.

The rules for converting to clausal form can be expressed more simply, to begin with, if implications and equivalences are reexpressed in terms of negation, conjunction and disjunction by using the equivalences:

> [X -> Y] <-> ¬X V Y
> [X <-> Y] <-> [X -> Y] & [Y -> X] i.e.
> [X <-> Y] <-> [¬X V Y] & [¬Y V X]
> where X and Y are any formulae.

Once implications and equivalences have been rewritten, the rest of the conversion consists of

(1) moving negations inside the sentence past conjunctions, disjunctions and quantifiers, until they stand only in front of atomic formulae,

(2) moving disjunctions inside the sentence past conjunctions and quantifiers, until they connect only atoms or negated atoms,

(3) eliminating existential quantifiers and

(4) reexpressing disjunctions

$$A_1 \vee \ldots \vee A_m \vee \neg B_1 \ldots \vee \neg B_n$$

of atoms and their negations as clauses

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n.$$

Negations can be moved in front of atoms by repeatedly applying the following equivalences:

```
¬[X & Y] <-> ¬X V ¬Y
¬[X V Y] <-> ¬X & ¬Y
¬∃v X    <-> ∀v ¬X
¬∀v X    <-> ∃v ¬X
¬¬X      <-> X
where X and Y are any formulae
and v is any variable.
```

Disjunctions can be moved inside a sentence until they connect only atoms and their negations by using the equivalences:

```
X V [Y & Z] <-> [X V Y] & [X V Z]
X V ∃v Y    <-> ∃v [X V Y]
X V ∀v Y    <-> ∀v [X V Y]
where the variable v does not occur in X.
```

The commutativity of disjunction

```
X V Y <-> Y V X
```

is needed to justify the similar equivalences

```
[Y & Z] V X <-> [Y V X] & [Z V X]
∃v Y V X    <-> ∃v [Y V X]
∀v Y V X    <-> ∀v [Y V X]
where v does not occur in X.
```

The preceding equivalences are sufficient to transform any sentence without quantifiers in standard form into an equivalent one in clausal form. The elimination of an existential quantifier, however, produces a sentence which is not equivalent. It introduces a constant or function symbol in order to name an individual which is referred to only implicitly in the original sentence. The new sentence implies, but is not implied by, the original sentence. Nevertheless, the elimination of the existential quantifier does not affect the consistency of the set of sentences as a whole.

Given a conjunction (or set) of sentences S, in order to eliminate existential quantifiers from S it is necessary to eliminate them from <u>sentences</u> of the form

| |
|---|
| $\forall v_1 \forall v_2 \ldots \forall v_n \exists u$ X |

belonging to S. Such a sentence can be replaced by the new sentence

| |
|---|
| $\forall v_1 \forall v_2 \ldots \forall v_n$ X'<br>where X' is obtained from X by replacing all free occurrences of u in X by the term $f(v_1,\ldots,v_n)$ where f is a function symbol which does not occur in S. |

If n=0 the term $f(v_1,\ldots,v_n)$ reduces to a constant symbol. Note that the replacement is not an equivalence and it only applies to sentences, not to formulae. The new conjunction (or set of sentences) is consistent (or inconsistent) if and only if S is.

In order to transform sentences belonging to S into the correct form, it is useful to move universal quantifiers inside conjunctions.

| |
|---|
| $\forall v$ [X & Y] <-> $\forall v$ X & $\forall v$ Y |

Repeated application of the preceding rules will convert any conjunction (or set) of sentences in standard form into a conjunction (or set) of sentences, each of which has a form

$$\forall v_1 \ldots \forall v_k\ [A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n]$$

which is equivalent to a clause

$$A_1,\ldots,A_m \leftarrow B_1,\ldots,B_n.$$

The preceding rules express the logic of a family of algorithms for converting from standard form to clausal form. All non-determinism$_1$ is of the don't care variety. An efficient algorithm is obtained by always applying the rules to an outermost propositional connective or quantifier, replacing the formula on the left hand side of an equivalence by the formula on the right hand side. Moreover, it is more convenient in practice to leave the implication sign intact and to apply derived equivalences. The following derived equivalences (see exercise 2) are the most useful.

| | | |
|---|---|---|
| [X -> Y & Z] | <-> | [X -> Y] & [X -> Z] |
| [X V Y -> Z] | <-> | [X -> Z] & [Y -> Z] |
| [X & ¬Y -> Z] | <-> | [X -> Y V Z] |
| [X -> ¬Y V Z] | <-> | [X & Y -> Z] |
| [X -> [Y -> Z]] | <-> | [X & Y -> Z] |
| [[X -> Y] -> Z] | <-> | [X V Z] & [Y -> Z] |
| X -> ∀v Y | <-> | ∀v [X -> Y] |
| X -> ∃v Y | <-> | ∃v [X -> Y] |
| ∀v Y -> X | <-> | ∃v [Y -> X] |
| ∃v Y -> X | <-> | ∀v [Y -> X] |
| where v does not occur in X. | | |

In addition, generalisations of the equivalences:

| | | |
|---|---|---|
| [U & [X V Y] -> Z] | <-> | [U & X -> Z] & [U & Y -> Z] |
| [U & [X -> Y] -> Z] | <-> | [U -> X V Z] & [U & Y -> Z] |

for example, are often useful as well. In order to apply them may require application of the commutativity of conjunction:

X & Y <-> Y & X

## Comparison of clausal form with standard form

Clausal form is a restricted subset of standard form. It has the advantage that simple, efficient, and reasonably natural resolution theorem provers have been developed for it. Standard form, however, allows more liberal means of expression. Some kinds of sentences can be expressed more economically and others more naturally than in clausal form. The analysis in the next few sections, of the cases in which standard form provides greater expressive power than clausal form, suggests that what is needed is not full unrestricted standard form but a limited extension of clausal form. In most cases it suffices to allow non-atomic formulae as conditions and conclusions of implications.

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$$

It is useful, in particular, to allow conclusions $A_i$ which are conjunctions of atoms and conditions $B_j$ which are implications. In addition it is useful to employ equivalences <-> for definitions instead of writing the two halves separately.

The ideal system of logic would combine the advantages of clausal form with those of standard form. In order to do so, it would need both to reduce to resolution for sentences already in clausal form and to resemble the natural deduction systems of Bledsoe [1971], Brown [1977], Bibel and Schreiber [1975], and Nevins [1974]. Such a system might result from combining the resolution rule with the rules which convert sentences from standard form to clausal form.

The satisfactory solution of the problem of deriving Horn clause programs from program specifications in standard form requires such a proof procedure. The problem has been investigated by Bibel [1976a, 1976b, 1978], Clark and Sickel [1977], and Hogger [1978a, 1978b, 1979]. Their derivation rules resemble both the rules for converting to clausal form as well as the resolution rule which behaves as procedure invocation. Proof procedures for the standard form of logic, which have some of the necessary properties, have been developed by Murray [1978] and by Manna and Waldinger [1978].

In the following sections we investigate a number of examples which illustrate the limitations of clausal form and the inadequacy of dealing with standard form simply by converting to clausal form and applying resolution. At the end of the chapter we shall consider the problem of deriving Horn clause programs from non-clausal specifications.

## Conjunctive conclusions and disjunctive conditions

Standard form is more economical than clausal form when the same conditions imply several conclusions or when the same conclusion is implied by alternative conditions.

Example Everyone makes mistakes.

Standard form $\forall x \exists y\ [\text{Human}(x) \rightarrow \text{Does}(x,y)\ \&\ \text{Mistake}(y)]$

Conversion (a) $\text{Human}(x) \rightarrow \text{Does}(x, m(x))\ \&\ \text{Mistake}(m(x))$

(b) $\neg\text{Human}(x) \lor [\text{Does}(x, m(x))\ \&\ \text{Mistake}(m(x))]$

(c) $[\neg\text{Human}(x) \lor \text{Does}(x, m(x))]\ \&$
$[\neg\text{Human}(x) \lor \text{Mistake}(m(x))]$

Clausal form (d) $\text{Does}(x, m(x)) \leftarrow \text{Human}(x)$
$\text{Mistake}(m(x)) \leftarrow \text{Human}(x)$

In the clausal form, the same condition Human(x) needs to be repeated for each separate conclusion. Notice that using the derived conversion rules for implication, the conversion from (a) to (d) can be done in one step.

Example One person is an ancestor of another if he is a parent of the other or he is an ancestor of an ancestor of the other.

Standard Form $\text{Anc}(x,y) \leftarrow \text{Par}(x,y) \lor \exists z\ [\text{Anc}(x,z)\ \&\ \text{Anc}(z,y)]$

Conversion (a) $\text{Anc}(x,y) \lor \neg[\text{Par}(x,y) \lor \exists z\ [\text{Anc}(x,z)\ \&\ \text{Anc}(z,y)]]$

(b) $\text{Anc}(x,y) \lor [\neg\text{Par}(x,y)\ \&\ \neg\exists z\ [\text{Anc}(x,z)\ \&\ \text{Anc}(z,y)]]$

(c) $[\text{Anc}(x,y) \lor \neg\text{Par}(x,y)]\ \&$
$[\text{Anc}(x,y) \lor \neg\exists z\ [\text{Anc}(x,z)\ \&\ \text{Anc}(z,y)]]$

(d) $[\text{Anc}(x,y) \lor \neg\text{Par}(x,y)]\ \&$
$[\text{Anc}(x,y) \lor \forall z\ [\neg\text{Anc}(x,z) \lor \neg\text{Anc}(z,y)]]$

(e) $[\text{Anc}(x,y) \lor \neg\text{Par}(x,y)]\ \&$
$\forall z\ [\text{Anc}(x,y) \lor \neg\text{Anc}(x,z) \lor \neg\text{Anc}(z,y)]$

Clausal form (f) $\text{Anc}(x,y) \leftarrow \text{Par}(x,y)$
$\text{Anc}(x,y) \leftarrow \text{Anc}(x,z), \text{Anc}(z,y)$

In the clausal form, the same conclusion needs to be repeated for each alternative condition. The conversion from standard form is simplified if the derived equvalences are used:

(a') $[\text{Anc}(x,y) \leftarrow \text{Par}(x,y)]\ \&$
$[\text{Anc}(x,y) \leftarrow \exists z\ [\text{Anc}(x,z)\ \&\ \text{Anc}(z,y)]]$

(b') $[\text{Anc}(x,y) \leftarrow \text{Par}(x,y)]\ \&$
$\forall z\ [\text{Anc}(x,y) \leftarrow \text{Anc}(x,z)\ \&\ \text{Anc}(z,y)]$

(c') $\text{Anc}(x,y) \leftarrow \text{Par}(x,y)$
$\text{Anc}(x,y) \leftarrow \text{Anc}(x,z), \text{Anc}(z,y)$

For the sake of simplicity we shall use the derived equivalences in the rest of the chapter.

## Disjunctive conclusions

Standard form is both more economical and more intelligible when the alternatives in a conclusion are conjunctions.

Example: The earth is round and finite or flat and infinite.

Standard form:

```
[Round(E) & Finite(E)] V [Flat(E) & Infinite(E)]
```

Conversion:

```
(a) [[Round(E) & Finite(E)] V Flat(E)] &
    [[Round(E) & Finite(E)] V Infinite(E)]

(b) [Round(E)  V Flat(E)] &
    [Finite(E) V Flat(E)] &
    [Round(E)  V Infinite(E)] &
    [Finite(E) V Infinite(E)]
```

Clausal form:

```
Round(E),  Flat(E) <-
Finite(E), Flat(E) <-
Round(E),  Infinite(E) <-
Finite(E), Infinite(E) <-
```

## Only-if halves of definitions

We shall argue in the next chapter that Horn clauses often express only the if-half of an if-and-only-if definition. The full if-and-only-if definition can be expressed compactly in the standard form by using the sign of equivalence <->. In the clausal form, the if-half and the only-if half need to be expressed separately. The only-if half generally expresses alternative conclusions and can be both uneconomical and unnatural.

Example: The only-if half of the if-and-only-if definition of ancestor.

Standard form:

```
Anc(x,y) -> Par(x,y) V ∃z[Anc(x,z) & Anc(z,y)]
```

Conversion:

```
(a) ∃z [Anc(x,y) -> Par(x,y) V [Anc(x,z) & Anc(z,y)]]

(b) Anc(x,y) -> Par(x,y) V
                [Anc(x, f(x,y)) & Anc(f(x,y), y)]

(c) Anc(x,y) -> [Par(x,y) V Anc(x, f(x,y))] &
                [Par(x,y) V Anc(f(x,y), y)]
```

Clausal form:

```
Par(x,y), Anc(x, f(x,y)) <- Anc(x,y)
Par(x,y), Anc(f(x,y), y) <- Anc(x,y)
```

## Implications as conditions of implications

It is common for sentences of natural language to have conditions which are themselves implications rather than simple atoms. Such sentences can be expressed directly and naturally in standard form, but may be difficult to understand in clausal form.

Example x⊃y is true if y is true whenever x is true.

Standard form True(x⊃y) <- [True(y) <- True(x)]

Clausal form True(x⊃y), True(x) <-
True(x⊃y) <- True(y)

Example Bob is happy if all his students like logic.

Standard form Happy(Bob) <- ∀x [Studentof(Bob,x) -> Likes(x,logic)]

Conversion (a) ∃x [Happy(Bob) <- [Studentof(Bob,x) ->
Likes(x,logic)]

(b) Happy(Bob) <- [Studentof(Bob,☺) -> Likes(☺,logic)]

Clausal form Happy(Bob), Studentof(Bob,☺) <-
Happy(Bob) <- Likes(☺,logic)

Example A supplier is preferred if all the parts he supplies arrive on time.

Standard form Preferred(x) <- Supplier(x) &
∀u [Supplies(x,u) -> Arriveontime(u)]

Clausal form Preferred(x) <- Supplier(x), Arriveontime(p(x))
Preferred(x), Supplies(x, p(x)) <- Supplier(x)

Example A set is well-ordered if and only if every non-empty subset has a least element. A set is non-empty if and only if it has at least one element. An element of a set is a least element if and only if it is less than or equal to every element of the set.

Standard form Wellordered(x) <-> ∀z [Hasleastelmt(z) <- z∈x &
Nonempty(z)]

Nonempty(z) <-> ∃u u∈z

Hasleastelmt(z) <-> ∃u [u∈z & ∀v [v∈z -> u≤v]]

Clausal form Wellordered(x), arb(x) ∈ x <-
Wellordered(x), Nonempty(arb(x)) <-
Wellordered(x) <- Hasleastelmt(arb(x))
Hasleastelmt(z) <- Wellordered(x), z∈x, Nonempty(z)
Nonempty(z) ∈ <- u∈z
select(z) ∈ z <- Nonempty(z)
Hasleastelmt(z), el(z,u) ∈ z <- u∈z
Hasleastelmt(z) <- u ≤ el(z,u), u∈z
smallest(z) ∈ z <- Hasleastelmt(z)
smallest(z) ≤ u <- Hasleastelmt(z), u∈z

Derivation of programs from specifications

Programs can be expressed more naturally in logic if implications are allowed as conditions. The definition of subset is a simple example:

$$x \subseteq y \leftarrow \forall z\ [z \in x \rightarrow z \in y]$$

The condition that "every element of x is an element of y" is neutral about the manner in which the elements of x are investigated and shown to be elements of y. In particular, it is consistent with the possibility that all elements of x are investigated simultaneously, in parallel. Such high-level specification is not possible in normal programming languages. It is not even possible with Horn clauses.

Suppose that sets are represented by finite lists. Then the notions of both membership and subset can be defined recursively by means of Horn clauses:

$$
\begin{array}{l}
z \in z.v \leftarrow \\
z \in u.v \leftarrow z \in v \\
nil \subseteq y \leftarrow \\
u.v \subseteq y \leftarrow u \in y,\ v \subseteq y
\end{array}
$$

The Horn clause program is less natural and closer to the level of the computer than the specification in standard form. It expresses details which are left to the initiative of the theorem prover in the standard form specification. It works, moreover, only for finite sets represented by means of lists. The standard form specification, on the other hand, works for both finite and infinite lists. Exercise (6b) demonstrates this for the notion of ordered list.

The use of logic is more widely accepted as a specification language than it is as a programming language. Methods for verifying conventional programs relative to logic specifications are complicated therefore by the need to relate two different languages. The methods of Floyd [1967], Manna [1969], Hoare [1969] and Dijkstra [1976] express specifications in logic and relate them to programs by defining the semantics of programs in logic.

Verification of programs is significantly easier when programs and specifications are expressed in the same language. This is confirmed by the results of Boyer and Moore [1975] who use LISP for both programs and specifications, Manna and Waldinger [1977], who use LISP for programs and LISP augmented with universally quantified implications for specifications, and Burstall and Darlington [1977], who use recursion equations for both programs and specifications. More recently, using the procedural interpretation of Horn clauses, deduction strategies for deriving logic programs from logic specifications have been developed by Clark and Tarnlund [1977], Bibel [1976a, 1976b, 1978], Clark and Sickel [1977], Hogger [1978a, 1978b, 1979] and Clark and Darlington [1978]. In addition, Manna and Waldinger [1978] have developed an extension of resolution for deriving LISP programs from logic specifications.

The derivation of logic programs from logic specifications has the special characteristic that deduction is used both to run programs and to derive programs from specifications. Programs can be regarded as computationally useful logical consequences of the specifications.

We shall illustrate the general method by deriving the Horn clause program for subset from the standard form specification The inference steps can be thought of as combining resolution with conversion to clausal form. We start with the if-and-only-if specifications of the subset and membership relations.

S1 $\quad x \subseteq y \leftrightarrow \forall z\ [\underline{z \in x} \rightarrow z \in y]$

S2 $\quad \forall z\ \neg[z \in nil] \qquad$ (i.e. $\leftarrow z \in nil$)

S3 $\quad \underline{z \in u.v} \leftrightarrow z=u \vee z \in v$

The basis of the recursive Horn clause program

$$nil \subseteq y \leftarrow$$

can be obtained directly by resolving the clausal form of S2 with the first of the two clauses

$$x \subseteq y,\ arb(x,y) \in x \leftarrow$$
$$x \subseteq y \leftarrow arb(x,y) \in y$$

obtained by converting S1 into clausal form.

The recursive clause of the program can be derived more naturally by reasoning with the specifications in standard form. By matching the underlined atoms in S1 and S3 we obtain

S4 $\quad u.v \subseteq y \leftarrow \forall z\ [[z=u \vee z \in v] \rightarrow z \in y].$

It suffices, in this case, to use only the if-half of the definition of subset. We can think of S4 as obtained by letting x be u.v in S1 and then using the equivalence S3 to replace $z \in u.v$ by $z=u \vee z \in v$. Next, we begin to convert S4 to clausal form.

S5 $\quad u.v \subseteq y \leftarrow \forall z\ [z=u \rightarrow z \in y]\ \&$
$\qquad\qquad\qquad\qquad \forall z\ [z \in v \rightarrow z \in y]$

Any further conversion would result in non-Horn clauses. Fortunately the two non-atomic conditions in S5 can be replaced by equivalent atomic ones.

S6 $\quad \forall z\ [z=u \rightarrow z \in y] \leftrightarrow u \in y$

S7 $\quad \forall z\ [z \in v \rightarrow z \in y] \leftrightarrow v \subseteq y$

Applying the two equivalences to S5 we obtain the rest of the program

$$u.v \subseteq y \leftarrow u \in y,\ v \subseteq y$$

It remains to demonstrate the equivalences S6 and S7. The second one S7 is easy; it is an instance of S1. The first equivalence is a special case of a more general equivalence

$\forall z\ [z=u \rightarrow X] \leftrightarrow X'$
where $X'$ is obtained from X by
replacing all occurrences of z by u.

which is useful in general.

The derivation of the subset program illustrates the use of inference rules which apply directly to the standard form and which resemble both resolution and the rules for converting from standard to clausal form.

## Exercises

1) Express the following sentences in standard form and transform them into clausal form.

a) A number is the maximum of a set of numbers if it belongs to the set and is $\geq$ all numbers which belong to the set. (Hint: Define an auxiliary relationship Dominates(x,y) which holds when x $\geq$ all numbers which belong to the set of numbers y.)

b) A list of numbers is ordered if it is empty or its first number is $\leq$ all numbers in the rest of the list and the rest of the list is ordered.

c) A number is the greatest common divisor of numbers x and y if it divides x and y and is $\geq$ all numbers which divide x and y.

2) The derived equivalences on page 199 can be justified by converting each half of an equivalence to the same formula, by replacing subformulae by equivalent subformulae. For example, both halves of the equivalence

X -> [Y & Z] <-> [X -> Y] & [X -> Z]

convert to the same formula

[¬ X V Y] & [¬ X V Z].

Derive the remaining equivalences on page 199.

3) a) Express the following assumptions in standard form and transform them into clausal form.

A dragon is happy if all its children can fly.
Green dragons can fly.
A dragon is green if at least one of its parents is green and is pink otherwise.

b) Use resolution (and factoring if necessary) to show:

(i) Green dragons are happy.
(ii) Childless dragons are happy.

You will need to supply some "obvious" missing assumptions.

c) What should a pink dragon do to be happy?

4) This exercise is an extension of exercise 8 of Chapter 2. Given data in the Supplier, Part and Supply tables, express the following queries in standard form. Use both the binary and n-ary representations.

a) What are the numbers of suppliers who supply all parts?

b) What are the names of suppliers who do not supply books?

c) What are the numbers of those suppliers who supply at least all parts supplied by John?

5) a) Express the following assumption in standard form and transform it into clausal form.

A logician is happy if all his arguments are sound.

b) Use resolution to show that the following conclusions are implied by the assumption.

(i) A Logician is happy if everyone's arguments are sound.
(ii) A logician is happy if he doesn't argue.

6) a) Express the following assumptions in standard form and transform them into clausal form.

(i) A sequence z is ordered if for every x, y, i and j,
x is the i-th element of z,
y is the j-th element of z and
$i \leq j$ imply $x \leq y$.

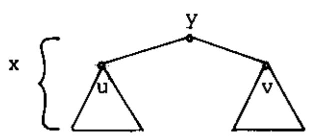
(ii) If $i \leq j$ then $u*i \leq u*j$, for all i, j and u.

(iii) The i-th element of sequence S is 3*i for all i.

b) Use resolution to show that the sequence S is ordered. Notice that S might have infinitely many elements.

7) Assume that the following relations are already defined:

$x \leq y$
$x > y$
Empty(x) the tree x contains no nodes.
Split(x,y,u,v) the tree x has root node labelled by item y, left subtree u and right subtree v.



a) Express the following definition of the relation Ord(x) in standard form:

The tree x is ordered if for every non-empty subtree z of x

i) all items which belong to the left subtree of z are ≤ the item at the root of z and

ii) all items which belong to the right subtree of z are > the item at the root of z.

You should define the following relations for this purpose.

```
Subtree(z,x)      z is a subtree of x
Belongs(y,x)      the item y belongs to tree x.
```

b) Transform the definition of Ord(x) into clausal form.

8) The relationship Sl(x,y), i.e. x is a sublist of y, can be specified by:

```
Sl(x,y)        <-> ∃u∃v∃w[Append(u,x,v) & Append(v,w,y)]
Append(x,y,z)  <-> [x=nil & y=z] V
                   ∃u∃v∃w[x=u.v & z=u.w & Append(v,y,w)]
```

Derive a recursive program for Sl(x,y), not involving Append, using the following assumptions about equality if necessary:

```
x.y = u.v <-> x=u & y=v
¬ ∃u∃v u.v = nil
x = x
```

9) The relationship Fact*(x,y,u,v) can be specified by

```
Fact*(x,y,u,v) <-> [Fact(x,y) -> Fact(u,v)]
Fact(x,y)      <-> [Zero(x) & Succ(x,y)]  V
                   ∃u∃v[Succ(u,x) & Fact'(u,v)
                                  & Times(x,v,y)]
Zero(0) <-
Succ(x, s(x)) <-
```

a) Derive a recursive program for Fact*(x,y,u,v), not involving Fact.

b) Show that Fact(u,v) <-> Fact*(0,s(0),u,v).

10) Given the specification

```
Ord(x) <-> ∀u∀v[Consec(u,v,x) -> u≤v]
```

derive a Horn clause program for Ord(x), using the following assumptions:

$$\neg\, Consec(u, v, nil)$$
$$\neg\, Consec(u, v, x.nil)$$
$$Consec(u, v, x.y) \leftrightarrow Consec(u,v,y) \lor \exists z[u=x \;\&\; y=v.z]$$