J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

HORN CLAUSES

This article introduces Horn clause logic, a subset, yet expressive, form of first-order logic. The syntax of Horn clauses along with the semantics and the related inference rule are presented. Background on how Horn clause logic is related to general first-order logic is covered. The subject matter is informally introduced through an example, which is then followed by a more rigorous description with additional examples. Horn clauses, named after Alfred Horn, constitute a language subset of first-order clauses, which represents a canonical, normal form for expressing first-order predicate calculus formulas. Although Horn clauses are syntactically restricted clauses, they provide sufficient expressiveness and semantics that are easier to implement on several computing platforms and are central to the study of logic programming languages and knowledge-based systems. In the early 1970s, through research and experiments with various formulations of problem-solving tasks, Horn clauses were found to be very useful and expressive (1). Lloyd (2) shows that all computable functions can be expressed in terms of a formal deduction system, the inference rule of which is SLD resolution and the language syntax of which consists of definite program clauses, a subset of Horn clauses. *SLD* resolution stands for a restricted resolution inference rule with a Selection function restricted to Linear resolution and Definite clauses. The history of the naming and definition of this inference rule is discussed in detail in this article.

Horn Clause Syntax and Terminology

Before proceeding with an introductory Horn clause example, some essential definitions are informally given. A clause is often written in the form

$$\forall x_1 \cdots \forall x_z A_1 \vee A_2 \vee \cdots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n$$

where $m \geq 0$, $n \geq 0$, $\vee$ is logical-OR, $\wedge$ is logical-AND, $\leftarrow$ is logical-IMPLIES, the A_i's and B_j's are atoms, $x_1, \ldots, x_z$ are all the variables that occur in A_i's and B_j's, and $\forall$ represents universal quantification (for all). An *atom* has the form of $P(t_1, t_2, \ldots, t_n)$, where P is a predicate symbol and t_i's are terms. A *term* is a constant, variable, or a function f applied to terms, t_i, notated as $f(t_1, t_2, \ldots, t_n)$. Terms with no variables are called *ground terms*. Similarly, atoms with only ground terms are called *ground atoms*.

The above clause is read as: "A_1 or A_2 or $\ldots$ or A_m is/are true if B_1 and B_2 and $\ldots$ B_n is/are true." Note that a clause may be empty, that is, $m = 0$ and $n = 0$; the truth assignment for an empty clause is always false and is considered to represent logic contradiction. The empty clause is denoted as $\square$.

When formally describing semantics for clauses it is convenient to write them using alternative syntax, but logically equivalent forms. One form is

$$\forall x_1 \cdots \forall x_z A_1 \vee A_2 \vee \cdots \vee A_m \vee \neg B_1 \vee \neg B_2 \vee \cdots \vee \neg B_n$$

The ¬ represents logical-NOT. Both negated atoms and nonnegated atoms are called negative and positive literals, respectively. The notation $|L|$ represents the atom in a literal with the possible negation symbol removed.

For the purpose of defining resolution and other related concepts, it is convenient to represent a clause as a set of literals with universal quantification of variables assumed:

$$\{A_1, A_2, \ldots, A_m, \neg B_1, \neg B_2, \ldots, \neg B_n\}$$

A *Horn clause* has at most one atom to the left of the implication arrow or, equivalently, at most one positive literal; that is, $0 \leq m \leq 1$. If $m = 1$, a Horn clause has the form

$$A \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n$$

The above is read as: "A is true if B_1 and B_2 and ... and B_n is/are true." This type of Horn clause is called a *definite program clause* in logic programming contexts. A is called the *head* of the clause and the B_i's collectively are called the *body* of the clause. If $n = 0$, the clause represents a statement that atom A is unconditionally true, that is, a *fact.* A fact is also called a *unit clause,* a special kind of definite program clause. Unit clauses are often written with the implication arrow omitted. A *Horn program* is a collection of definite program clauses. If $m = 0$ and $n \geq 1$, the clause is called a *definite goal clause,* a *query,* or, simply a *goal clause* in logic programming contexts.

In the following examples, strings composed of lowercase letters and Arabic digits represent constants and upper case letters represent variables. Also, the logical-AND operator ($\wedge$) is replaced by a comma (,). Truth values are represented as TRUE and FALSE. (Note that when clauses are represented in set notation, the comma delimits the literals and no longer represents logical-AND.)

Relationship Horn Program Example. The following example illustrates the use of Horn clauses for expressing the relationships of father to child. The atom *Father*(F, C) represents the statement that F is a father of child C and *Parent*(F, C) represents the statement "F is a parent of C."

Father(F, C) ← *Parent*(F, C), *Male*(F)
Parent(*joe, dave*)
Parent(*joe, john*)
Parent(*mary, john*)
Male(*joe*)
Male(*john*)
Male(*dave*)
Male(*mark*)
Female(*mary*)

Informal Horn Clause Semantics

Informal Model-Theoretic Semantics of Relationship Example. The first clause is read as: "F is a father of C if F is a parent of C and F is male." The remaining clauses are facts, each assumed to be true assertions. For example, *Parent*(*joe, dave*) asserts that "joe is a parent of dave." Informally, the *Parent* predicate symbol is intended to represent a parent relation over a set of people, called the universe of discourse. The universe of discourse might represent the set of people in a particular country, state, organization, etc.

The assignment of terms to elements in the universe of discourse and the assignment of relations to predicate symbols is an interpretation. The set of names {*joe, dave, john, mark, mary*} is intended to represent particular people in a universe of discourse; the representation is an interpretation. The binary relation {(*joe,*

dave), (*joe, john*), (*mary, john*)} is assigned to predicate symbol *Parent.* The set {*joe, john, dave, mark*} is a unary relation assigned to predicate symbol *Male.* The set {*mary*} is assigned to predicate symbol *Female.* These predicate symbol assignments complete the interpretation.

By substituting the variables with all possible name values chosen from a universe of names, one is able to "compute" the relation *Father* by noting which names satisfy or make the rule TRUE according to the usual semantics associated with logical-AND and logical-IMPLIES semantics. For example, given the substitutions that replace F with *joe* and C with *john,* the rule is TRUE; so the tuple (*joe, john*) is in the relation assigned to *Father.* However, when any name other than *joe* is assigned to variable F, the rule is not necessarily TRUE since either the name is not a left-hand member of the *Parent* relation or is not in the *Male* unary relation, as assigned in the interpretation. Choosing the universe of names, assigning the relations to all predicate symbols (even those occurring only in the head of a Horn clause), and evaluating the truth values of each clause is one approach to giving a semantics to sets of clauses and Horn clauses in general. Each universe and assignment comprises an interpretation. The interpretations that satisfy or make all clauses true for all substitutions of variables are models.

Informal Procedural Semantics of Relationship Example. Procedural semantics are based on a formal deduction system, which comprises a language (Horn clauses in this case), a set of proper axioms (the set of clauses), and an inference rule. The procedural semantics describe how to construct proofs. Constructing a proof involves applying an inference rule to a set of clausal axioms to deduce more clauses, which, in turn, are added to the set of axioms. A sequence of inferences starting with a set of axioms is a proof of the last clause in the sequence.

Logicians are interested in inference rules that are sound and complete. An inference rule is characterized as being *sound* if and only if it yields only clauses that are satisfied by the model-theoretic semantics, that is, they are TRUE in every model of the theory. An inference rule is *complete* if and only if a proof can be constructed for any clause satisfied by the model-theoretic semantics.

Resolution is a sound and complete inference rule for general clauses. Readers interested in further details should consult the original work on resolution by Robinson (3) or textbooks such as those by Chang and Lee (4) or Loveland (5). Resolution and related concepts are briefly introduced later in this article for the purpose of providing a context for SLD resolution, which is a more restricted resolution inference rule for Horn clauses.

Informally, resolution involves two general steps. The first step involves unification of two atoms, each belonging to two distinct clauses. One of the atoms appears negated in a clause and the other occurs as a positive literal in the other clause. Unification is a substitution or an assignment of variables to (not necessarily ground) terms that make the atoms syntactically match. A substitution of a variable X with a term t is often denoted X/t. A unifier is denoted as a set of these variable assignments.

The second step involves applying the substitution for all variables in each clause and canceling these matching atoms and leaving the remaining atoms. The remaining atoms form a new clause, called the *resolvent,* that is added to the set of clauses that potentially could be used in a resolution step.

A proof in a resolution-based deduction system is one based on reaching a contradiction—a proof by contradiction. In a system composed only of Horn clauses, one begins with the goal or query clause and, through resolution at each step, attempts to arrive at a contradiction, which is represented by the empty clause, $\square$. The sequence of steps that begins with a goal clause and that arrives at a contradiction is called a *refutation.* Because there may be several choices in clauses to use in a resolution step, the construction of a refutation is tantamount to a search problem.

The preceding description to finding a refutation is called a top-down procedure. In contrast there is a bottom-up approach that starts with the unit (fact) clauses and works "up" to the goal clause. For this example, we utilize the top-down procedure to illustrate a refutation. We give a more formal description later. For now, we give an overview.

4 HORN CLAUSES

Suppose that we attempt to prove that "joe is a father of dave." Toward a contradiction, one negates the ground atom *Father*(*joe, dave*) and expresses it as a goal clause:

$$\neg Father(joe, dave)$$

This unifies with the head of the Father rule with unifier {*F*/*joe*, *C*/*dave*}. Applying this unifier to the other literals in the Father rule, we are left with the resolvent:

$$\neg Parent(joe, dave) \vee \neg Male(joe)$$

This clause is now added to the collection of clauses. In general resolution-based deduction systems, there are many choices of clauses to use in each resolution step and, moreover, there may be more than one choice in the literals that unifies. The choices implemented in such a reasoning system define the operational or procedural semantics. In Horn clause–based deduction systems, the computational model or operational semantics requires that one of the clauses to be used in each resolution step be the most recently added goal clause. This strategy is one component of SLD resolution, a restricted resolution rule used for Horn clause–based systems. Using this strategy, we continue with the proof example.

Although we require that one of the clauses be a goal clause, we need to select a literal in a goal clause. Defining a rule or function that selects a literal in a clause is the other component of SLD resolution.

In this example, we choose the leftmost literal ¬*Parent*(*joe, dave*) instead of ¬*Male*(*joe*). The choice is arbitrary here. This unifies with the unit clause *Parent*(*joe, dave*) with the empty unifier, which requires no variable substitutions. We are left with the (goal clause) resolvent:

$$\neg Male(joe)$$

Finally, ¬*Male*(*joe*) unifies with the unit clause *Male*(*joe*) with the empty unifier. The resolvent is the empty clause, which represents contradiction. With this, we have shown a proof that *Father*(*joe, dave*) is TRUE. That is, we negated what we were trying to prove and, through a sequence of resolution steps, arrived at a contradiction. This sequence is a refutation.

Note that since one and only one of the clauses used in SLD resolution must be a goal clause, and a goal clause contains only negative literals, the negative literal used in a unification step must unify with the head of a definite clause. We do not need to look at the clause bodies when we apply the unification step. In general, there may be more than one clause head to unify with. Choosing which goal clause literal and then which definite clause head is specified in the computational model used to construct refutations. If at any step there is no match between the selected literal and any definite clause head, backtracking to the previous goal clause and a different program clause is chosen at that step. This process continues until a contradiction is reached or backtracking goes back to the original goal clause and we do not have more choices. In that case the operational semantics are defined to give an answer: no. Notice that there is no need to backtrack over the goal literal selected, since in order to obtain the empty clause all literals in the goal must be resolved away. If a refutation exists we will find one independently of the literal selected.

When variables are present in the original goal clause, the unifiers used at each resolution step are composed and the substitutions for the original goal clause are called the *computed answer substitutions*. Because there may be more than one clause to resolve with the goal clause at each step of a refutation, there is potentially more than one computed answer. How this is efficiently implemented is another issue in Horn clause–based logic programming systems, which is beyond the scope of this article.

For example, if the goal clause ¬*Father*(*X*, *dave*) is given, the computed answer would be {*X*/*joe*}. This would similarly follow the steps given in the previously shown refutation. If the goal clause ¬*Father*(*X*, *Y*) were given, two computed answers would be given:

(1) $\{X/joe, Y/dave\}$
(2) $\{X/joe, Y/john\}$.

In this second query, two refutations are constructed via SLD resolution and backtracking. The procedure of computing all possible answers, starting with the query, terminates when no more refutations can be constructed with SLD resolution.

A More Formal Semantics

The study of logic, including syntactic and semantic aspects, can be broadly viewed as one approach to reasoning about abstract and concrete entities and the relationships between these entities; that is, it is one approach to a knowledge representation scheme or paradigm.

One may classify a logic according to various attributes that reflect some assumptions made regarding the relationships between these entities within a universe of discourse that the logic represents. For example, one taxonomy of logic may be based upon the nature of the range of the variables, if any, of the logic language. Within a zero-order (propositional) logic language, there are no variables; the propositional symbols represent statements (propositions) about particular entities. Higher-order logics introduce variables that range not only over these universe entities, but also the relationships between the entities (second-order) the relationships among relationships between entities (third-order), etc. For each of the orders, each class of variables may be quantified. Horn clauses are a syntactically restricted form of first-order logic, the language of which includes variables that range over the fundamental entities or elements of the universe of discourse.

There is a multitude of other attributes that can be used to classify a logic. We state just a few here to indicate our assumptions. For example, one may utilize different underlying (possibly partially ordered) sets of truth values. Related to this is the classification of a logic based upon the assumption made about the "law of the excluded middle," which informally means that all statements in a logic are either true or false. Another classification can be based upon whether a logic is monotonic or not. Within a nonmonotonic logic the assignment of a truth value of one statement may alter the truth value of another; within a monotonic logic truth values are not altered. Only two-valued, monotonic logic and the law of the excluded middle are utilized within Horn clause logic.

There are two primary approaches to assigning semantics or meaning to sets of clauses and Horn clauses. These are model-theoretic and procedural semantics. Each of these are considered using the relationship program example as a means for discussion and presentation.

Model-Theoretic Semantics. The model-theoretic semantics of first-order logic defines an assignment of the truth values of TRUE or FALSE to every formula in the logic. This assignment is done in several steps. First, every constant in the logic is identified with an element in a particular domain, for example, the rational numbers. Then every function symbol in the logic is identified with a function over the same domain. Then the interpretation is extended to the predicate symbols by assigning to each predicate a relation, again using the same domain. Finally, based on these assignments, the truth value of formulas is defined inductively by first assigning truth values to atomic formulas and then to more complex formulas. In the case of Horn clauses this process can be highly simplified. Interpretations are restricted to a very special class called *Herbrand interpretations.* A Herbrand interpretation is simply a set of ground atomic formulas, that is, atomic formulas with no variables. The meaning that these interpretations assign to formulas is as follows. The truth value of any ground atomic formula in the set is TRUE, and for any other ground formula its truth value assignment if FALSE. Before we say how clauses are interpreted let us see an example.

From our family relationship program, a Herbrand interpretation could be the set $\{Father(mary, john), Parent(joe, david), Male(david), Father(mark, mary)\}$. Note that this interpretation is not a good interpretation

since the program does not seem to imply that *mary* is the father of *john*. Good interpretations are called Herbrand models, or simple models, and are defined as follows.

Let $Ground(\Phi)$ be the ground instances of all the clauses in a program Φ. That is, take any clause C in Φ and replace all the variables in C with ground terms; the resulting clause must be in $Ground(\Phi)$. A Herbrand interpretation M is said to be the intended model of the program φ if (1) for every clause in $Ground(\Phi)$ whose atoms in the body are in M, its head is also in M and (2) there is no proper subset of M with property (1).

Continuing with our family example, the set {*Parent*(*joe, dave*), *Parent*(*joe, john*), *Parent*(*mary, john*), *Male*(*joe*), *Male*(*john*), *Male*(*dave*), *Male*(*mark*), *Female*(*mary*), *Father*(*joe*, *dave*), *Father*(*joe*, *john*)} is the intended model of our program, and coincides with our intuition about the meaning of the program.

It can be shown that for any Horn program this model always exists and is unique. Let us look at another example.

$arc(a, b)$
$arc(b, c)$
$arc(b, d)$
$path(X, Y) \leftarrow arc(X, Y)$
$path(X, Y) \leftarrow path(X, Z), path(Z, Y)$

The program is intended to represent the arcs in a graph and paths between nodes in the graph. The meaning of the first three clauses is obvious. The last two clauses are more interesting. The fourth clause says that there is a path from X to Y if there is an arc from X to Y. The last rule says that there can also be a path from X to Y if there is a path from X to an intermediate point Z and then a path from Z to Y. With a careful analysis the reader can notice that the intended model for this program is

$$\{arc(a, b), arc(b, c), arc(b, d), path(a, b), path(a, c), path(a, d), path(b, c), path(b, d)\}$$

With the definition of the intended model of a Horn program we can define the *answers* to a goal. Recall that a goal clause is a clause of the form $\not B_1 \vee \cdots \vee \not B_n$, also written as $\leftarrow B_1, \ldots, B_n$. Answers are defined using substitutions. Formally, a substitution θ is an assignment of terms to variable names. A substitution is denoted as follows: $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$. The notation $E\theta$ indicates the application of a substitution θ to E, where E represents a term, atom, literal, clause, or set of clauses. $E\theta$ represents E with all variables replaced with the terms. For example, if $E = \{p(X,Y), q(X,Z)\}$ and θ $\{X/a, Y/Z\}$, then $E\theta = \{p(a, Z), q(a, Z)\}$. An answer for a goal clause G in a Horn program Φ is a substitution θ such that any ground instance of an atom in $G\theta$ is a member of the intended model of Φ. From our last example the goal $\leftarrow path(a,X)$ has three answers, $\theta_1 = \{X/b\}$, $\theta_2 = \{X/c\}$, and $\theta_3 = \{X/d\}$.

There are many properties connected Herbrand models and answers to first-order logic. Details can be found in Refs. 2 and 6.

Procedural Semantics. The following subsections present a general procedure to compute answers. First, unification and general resolution is presented to provide the context for a restricted resolution-based procedure, called the SLD resolution, which is central to the procedural semantics of Horn clauses.

Unification. A key component in the procedure to compute answers is called unification. Unification is the process of finding a substitution that makes two terms or atoms syntactically identical by applying the substitution to the terms or atoms.

There may be more than one unifier. There are some unifiers that have less term-to-variable assignments than others; the unifiers with less are referred to as the most general. Since any of them could be used as a general unifier, one is chosen as a representative and called the most general unifier (*mgu*).

Table 1. Unification Examples

	Expression 1 (E1)	Expression 2 (E2)	(an) mgu(E1, E2)
(1)	$f(X, a)$	$f(g(Y), Z)$	$\{X/g(Y), Z/a\}$
(2)	$f(X, a)$	$f(g(Y), b)$	fail
(3)	$p(f(X), X, g(b))$	$p(f(Y), Z, g(X))$	$\{X/b, Y/b, Z/b\}$
(4)	$p(f(X), X, g(b))$	$p(f(Y), Z, g(X_1))$	$\{X/Y, Z/Y, X_1/b\}$
(5)	$p(X, X)$	$p(Y, f(Z))$	$\{X/f(Z), Y/f(Z)\}$
(6)	$p(X, X)$	$p(Y, f(Y))$	fail

Given that E_1 and E_2 are both terms or atoms, hereafter called expressions, the notation mgu(E_1, E_2) represents an mgu of expressions E_1 and E_2. Shown in Table 1 are examples of applying the mgu function to expressions.

Properties about unification and an algorithm to compute most general unifiers can be found in Ref. 7.

Binary Resolution. The second component of the procedure to compute answers is *binary resolution.* The general resolution introduced by Robinson (8) is an inference rule that deals with general clauses and has two steps: binary resolution and factoring. Factoring is presented to be complete in the discussion; however, it is not necessary when using SLD resolution and Horn clauses.

Before describing binary resolution, some convenient terminology is introduced. Two clauses C_1 and C_2 are *renamed apart* when variables in the clauses are renamed so that the set of variables that appear in C_1 is disjoint from those appearing in C_2. If at least one variable is renamed in a clause C and the resultant clause is C', then C' is also called a *variant* of C.

Binary resolution can be defined as follows. Let C_1 and C_2 be two clauses renamed apart. A clause C is a *resolvent* of C_1 and C_2 if and only if

(1) there exists a literal $L_1 \in C_1$ and a literal $\not L_2 \in C_2$ such that $|L_1|$ and $|L_2|$ are unifiable with mgu($|L_1|$, $|L_2|$) $= \theta$, and
(2) $C = C_1\theta - \{L_1\theta\} \cup C_2\theta - \{L_2\theta\}$.

We say C_1 and C_2 are *parent* clauses of resolvent C. We also say that L_1 and L_2 are the *literals resolved upon.* An example of resolvents is as follows. Let

$$C_1 = \{p(X_1), \neg q(f(Y_1))\} \quad \text{and} \quad C_2 = \{\neg p(a), q(X_2), r(g(X_2))\}$$

C' and C'' are two possible resolvents of parents C_1 and C_2:

(1) Resolving upon literals $p(X_1)$ and $\not p(a)$ from clauses C_1 and C_2, respectively, we have

$$mgu(p(X_1), p(a)) = \{X_1/a\} \quad \text{and}$$
$$C' = \{\neg q(f(Y_1)), q(X_2), r(g(X_2))\}$$

(2) Resolving upon literals $q(f(Y_1))$ and $q(X_2)$ from clause C_1 and C_2, respectively, we have

$$mgu(q(f(Y_1)), q(X_2)) = \{X_2/f(Y_1))\} \quad \text{and}$$
$$C'' = \{p(X_1), p(a), r(g(f(Y_1)))\}$$

A refutation system that has only the binary resolution inference rule is not refutation complete. We consider the following classic example (see Ref. 8). Let

$$C_1 = \{p(X_1), p(Y_1)\} \quad \text{and} \quad C_2 = \{\neg p(X_2), \neg p(Y_2)\}$$

Clearly, the set $\Gamma = \{C_1, C_2\}$ is unsatisfiable; that is, it has no models. However, one needs more than just the binary resolution inference rule to refute Γ. All the resolvents of C_1 and C_2 are variants of the clause $\{p(X), p(Y)\}$.

Let $C = \{L_1, \ldots, L_n\}$ be a clause for which there exists two (same sign) literals L_i and L_j $(i = j)$ that are unifiable with a unifier θ. A *factor* of C is the clause $C\theta$.

In the following, the most general unifier will be used in factoring. We give three examples of factoring:

(1) Factoring C_1 and C_2 as given above, we get, respectively,

$$\{p(X)\} \quad \text{and} \quad \{\neg p(X)\}$$

Clearly, the resolvent of these two clauses is $\square$. Thus, through factoring and then the resolving of those factors, we have a refutation for $\Gamma = \{C_1, C_2\}$.

(2) $C'' = \{p(X_1), p(a), r(g(f(Y_1)))\}$ as shown above in the resolvent example can be factored as

$$\{p(a), r(g(f(Y_1)))\}$$

$C' = \{q(f(Y_1)), q(X_2), r(g(X_2))\}$ as shown above in the resolvent example cannot be factored.

(3) Suppose clause $C = \{p(X), q(X), q(f(Y))\}$. A factor of C is $\{p(f(Y)), q(f(Y))\}$.

Let Γ be a set of clauses. A *resolution deduction* (or *resolution derivation*) of clause C from Γ is a deduction $C_1, C_2, \ldots, C_n$ such that C is exactly C_n and each C_i $(1 \leq i \leq n)$ is a clause for which

(1) $C_i \in \Gamma$,

(2) C_i is a factor of a clause C_j $(j < i)$, or

(3) C_i is a resolvent of preceding clauses $C_1, C_2, \ldots, C_{i-1}$.

The resolution derivation is denoted: $\Gamma \vdash C$.

A *resolution refutation* of a set of clauses Γ is a resolution deduction of $\square$ from Γ and is denoted $\Gamma \vdash \square$.

Resolution with Horn Clauses. Since searching for refutations in a general resolution system can be quite expensive in time and space, naturally one investigates useful subclasses of clauses and restricted resolution inference rules. Kowalski and Kuehner (9) indicate that an inference system should satisfy the following criteria:

(1) "It should admit few redundant derivations and limit those which are irrelevant to the proof."

(2) "It should admit simple proofs."

(3) "It should determine a search space which is amenable to a variety of methods of heuristic searches."

It has been shown that SLD resolution meets the above criteria. The refutation procedure based upon SLD resolution was first described by Kowalski (10). As stated by Lloyd (2) along with Apt and van Emden

(11), the "SLD" in SLD resolution is an abbreviation for "linear resolution with a selection function for definite clauses." We note that there are other assumptions made about what "SLD" represents. In Ref. 12 Ringwood gives a brief history of the confusion about the SLD abbreviation; we leave it to the interested reader to consult that article for details.

SLD resolution is a further restriction to other restricted forms of resolution, namely linear-input resolution. Linear and input resolutions are briefly described as follows; a more detailed description can be found in Ref. 4.

- *Linear resolution* requires that both parents of a resolvent must be a previous resolvent; the first resolvent's parents, of course, must be the input clauses (which are the proper axioms).
- *Input resolution* requires that one of the two parents be an input clause. (The input clauses can be thought of as proper axioms.)
- *Linear-input resolution* naturally combines the previous two restrictions. It can be shown that linear-input resolution is only refutation complete for Horn clauses (12).

To describe SLD resolution, several definitions are presented first.

Let G be a goal clause $\leftarrow A_1, \ldots, A_i, \ldots, A_m$. Let C be a definite clause, $A \leftarrow B_1, \ldots, B_n$. A goal clause G', also called a resolvent, is derived from G and C using mgu θ if the following constraints hold:

(1) A_i is an atom from G called the selected atom.
(2) θ is the mgu of A_i and A, the head atom of clause C.
(3) G'' is the resolvent goal clause $(\leftarrow A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots A_m)\theta$.

An SLD derivation is a restricted resolution derivation as follows. Let Φ be a Horn program and G_0 a goal clause. An SLD derivation of $\Phi \cup \{G_0\}$ is a (not necessarily finite) sequence of

(1) Goals $G_0, G_1, \ldots$
(2) Definite clauses from Φ, $C_1, C_2, \ldots$
(3) mgus $\theta_1, \theta_2, \ldots$

such that G_{i+1} is derived from G_i and a (renamed apart from G_i) clause C_{i+1} using θ_{i+1}.

It should be noted that the variables in each C_{i+1} may need to be renamed so that there are no common names with the corresponding goal G_i. Such a renamed clause is also referred to as a variant of a clause. Each clause variant C_i is called an input clause of the derivation.

Let Φ be a definite program and G_0 a goal clause. A finite SLD derivation of $\Phi \cup \{G_0\}$, whose goal sequence is $G_0, G_1, \ldots, G_n = \square$, is called an SLD refutation of $\Phi \cup \{G_0\}$ of length n.

An SLD derivation, like all general resolution-based deductions, can be *finite* or *infinite.* For finite SLD derivations, they are partitioned into *successful* or *failed* ones. A *successful* SLD derivation is simply another name for an SLD refutation.

A *failed* SLD derivation is one in which the last clause in the derivation contains a selected atom that does not unify with any program clause head. It is also shown that SLD resolution is sound and complete with respect to Horn clause logic (2). The *success set* of a definite program Φ is the set of all ground atoms A such that $\Phi \cup \{\leftarrow A\}$ has an SLD refutation. It can be shown that the success set of a program Φ is exactly the intended Herbrand model of Φ.

Let Φ be a Horn program and G a goal clause. A *computed answer substitution* θ for $\Phi \cup \{G\}$ is the resultant substitution of restricting the composition of $\theta_1 \ldots \theta_n$ to the variables in the goal G, where $\theta_1 \ldots \theta_n$ is the sequence of mgu's used in an SLD refutation of $\Phi \cup \{G\}$.

Table 2. SLD Refutation with Ground Goal Clause Example

	Goal Clause	Definite Clause	mgu
G_0	← Father(joe, dave)		
	⇓	Father(F, C) ← Parent(F, C), Male(F)	θ_1 = {F/joe, C/dave}
G_1	← Parent(joe, dave), Male(joe)		
	⇓	Parent(joe, dave)	θ_2 = { }
G_2	← Male(joe)		
	⇓	Male(joe)	θ_3 = { }
G_3	□		

Examples of Refutations with the Relationship Horn Program. Let Φ be set of definite clauses that represent a family of people as given earlier.

Father(F, C) ← *Parent*(F, C), *Male*(F)
Parent(*joe, dave*)
Parent(*joe, john*)
Parent(*mary, john*)
Male(*joe*)
Male(*john*)
Male(*dave*)
Male(*mark*)
Female(*mary*)

We show an SLD refutation and illustrate how a computed answer is constructed. Suppose we wish to prove that "joe is a father of dave." One negates the ground atom *Father*(*joe, dave*) and expresses it as a goal clause:

$$G_0 = \neg Father(joe, dave)$$

A refutation for $\Phi \cup \{G_0\}$ is shown in Table 2. By constructing an SLD refutation, we have shown that "joe is a father of dave."

When implementing SLD resolution, one must commit to a selection function. This requires a well-defined procedure for selecting one and only one of the literals in the goal clause at each step in an SLD derivation (or refutation). In the above example, at step 1, the leftmost literal, ~~*Parent*~~(*joe, dave*) was chosen; however, ~~*Male*~~(*joe*) could have been chosen. It can be shown that if a refutation exists, it may be found, independent of the literal selected.

Another aspect of implementing SLD resolution involves the order of potential clause heads tried when attempting to unify the selected literal. This is where backtracking is employed. The reader should consult Ref. 2 for details and a starting point for investigating other sources that present machine implementation details.

Suppose one wants to compute the answers to "Who is the father of whom?" We illustrate this using the same set of definite clauses that represent a family of people. An example of computing one of the answers is shown in Table 3. Note that the definite clauses from Φ have to be renamed apart from the goal at some steps. The computed answer is {*F*/*joe*, *C*/*dave*}. That is, "joe is a father of dave" as shown earlier. This answer is constructed by composing the mgu's used at each step as follows:

$$\{F/joe, C/dave\} = \theta_1\theta_2\theta_3 = \{F/F_1, C/C_1\}\{F_1/joe, C_1/dave\}\{\}$$

Table 3. SLD Refutation and Computed Answer Example

	Goal Clause	Definite Clause	mgu
G_0	$\leftarrow$ Father(F, C)		
	$\Downarrow$	Father(F_1, C_1) $\leftarrow$ Parent(F_1, C_1), Male(F_1)	θ_1 = {F/F_1, C/C_1}
G_1	$\leftarrow$ Parent(F_1, C_1), Male(F_1)		
	$\Downarrow$	Parent(joe, dave)	θ_2 = {F_1/joe, C_1/dave}
G_2	$\leftarrow$ Male(joe)		
	$\Downarrow$	Male(joe)	θ_3 = { }
G_3	$\square$		

Table 4. Alternate SLD Refutation to Table 3 Example

	Goal Clause	Definite Clause	mgu
G_0	$\leftarrow$ Father(F, C)		
	$\Downarrow$	Father(F_1, C_1) $\leftarrow$ Parent(F_1, C_1), Male(F_1)	θ_1 = {F/F_1, C/C_1}
G_1	$\leftarrow$ Parent(F_1, C_1), Male(F_1)		
	$\Downarrow$	Parent(joe, john)	θ_2 = {F_1/joe, C_1/john}
G_2	$\leftarrow$ Male(joe)		
	$\Downarrow$	Male(joe)	θ_3 = { }
G_3	$\square$		

Yet another SLD refutation for the same program and goal clause is given in Table 4. This computed answer shows that "joe is a father of john." At step 1 the leftmost literal is chosen again, as given previously, but a different clause is used. If a different selection function were used, one may chose the rightmost literal at step 1. Careful analysis reveals that the same answers would be computed as the two preceding examples illustrate.

Data Structures in Horn Programs. In this section we will show how function symbols are used in Horn programs to build data structures. The following examples all implement different operations over lists of terms. A list of terms is a finite sequence of zero or more terms. Three examples of lists are: (a, *joe,* b), ($f(a)$, $g(a, a)$, a), and (b, b, b, b). We will use the following terms to represent lists:

(1) The special constant *nil* is a list, representing the empty list.
(2) If t is a term and L is a list then $list(t, L)$ is also a list.

The following program will test or define lists:

$$IsaList(nil)$$
$$IsaList(list(E, L)) \leftarrow IsaList(L)$$

The definition of first and last element of list is

$$First(list(F, L), F) \leftarrow IsaList(L)$$
$$Last(list(L, nil), L)$$
$$Last(list(E, Y), L) \leftarrow Last(L, Y)$$

Table 5. Representing Data Structures

Goal Clause	Definite Clause	mgu
$\leftarrow Last(list(a, list(c, nil))), X)$		
$\downarrow$	$Last(list(E_1, L_1), X_1 \leftarrow Last(L_1, X_1)$	$\sigma_1 = \{E_1/a, L_1/list(b, list(c, nil)), X_1/X\}$
$\leftarrow Last(list(b, list(c, nil)), X)$		
$\downarrow$	$Last(list(E_2, L_2), X_2 \leftarrow Last(L_2, X_2)$	$\sigma_2 = \{E_2/b, L_2/list(c, nil), X_2/X\}$
$\leftarrow Last(list(c, nil), X)$		
$\downarrow$	$Last(list(X_3, nil), X_3)$	$\sigma_3 = \{X_3/c, X/c\}$
$\square$		

Suppose one wants to compute the last element of the list (a, b, c). This is translated into the goal clause $\leftarrow Last(list(a, list(b, list(c, nil))), X)$. The refutation for this goal is shown in Table 5. The computed answer is $\{X/c\}$. Our last example shows the definition of member:

$$Member(E, list(E, L)) \leftarrow IsaList(L)$$
$$Member(E, list(X, L)) \leftarrow Member(E, L)$$

In this program if one asks the query $\leftarrow Member(E, list(a, list(b, list(c, nil))))$, there will be, as expected, three computed answer substitutions: $\{E/a\}$, $\{E/b\}$, and $\{E/c\}$.

Summary

The discovery of Horn clauses has made programming in logic a reality. Perhaps the most evident contribution to computer science is the logic programming language *Prolog* (1,13). Logic programming has matured into a field in its own right and has found applications in such diverse fields as program verification, computational biology, and databases. Deductive databases are a direct descendent of Horn logic and have had a major impact on database technology. There are two major conferences dedicated to the presentation of research work in the area: the *North American Conference on Logic Programming* and the *International Conference of Logic Programming.* There is also a journal, the *Journal of Logic Programming,* which recently celebrated its 25th anniversary. One can also regularly find papers in the major journals of databases, programming languages, and artificial intelligence founded in logic programming concepts.

Logic programs are not restricted to Horn clauses and most logic programming environments include full clausal syntax and many variations of semantics (14). Some of the variations include allowing negation of atoms in the body of a clause, allowing a disjunction of atoms in a rule head, adding constraints to clauses, and allowing concurrency to play an integral part of the implementation of constructing computed answers. We invite the reader to browse over the conference proceedings of the logic programming conferences to find more about the recent direction of the field or to visit the *Association of Logic Programming* web site at http://www.cwi.nl/projects/alp/.

BIBLIOGRAPHY

1. J. Cohen, A view of the origins and development of Prolog, *Commun. ACM*, **31** (1): 26–36, 1988.
2. J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed., New York: Springer-Verlag, 1987.
3. J. A. Robinson, A machine-oriented logic based on the resolution rule, *J. ACM*, **12** (1): 23–41, 1965.

4. C.-L. Chang, R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, New York: Academic Press, 1973.
5. D. Loveland, *Automated Theorem Proving: A Logical Basis*, New York: North-Holland, 1978.
6. K. R. Apt, Logic programming, in J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, New York: North-Holland, 1990, Vol. B, pp. 493–574.
7. J.-L. Lassez, M. J. Maher, K. Marriott, Unification Revisited, in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, San Mateo, CA: Morgan Kaufmann, 1988, pp. 587–626.
8. L. Wos, *et al.*, *Automated Reasoning: Introduction and Applications*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
9. R. Kowalski, D. Kuehner, Linear resolution with selection function, *Artif. Intell.*, **2**: 227–260, 1971.
10. R. Kowalski, Predicate Logic as a Programming Language, in *Proc. Inf. Proc. '74*, Stockholm: North-Holland, 1974, pp. 569–574.
11. K. R. Apt, M. H. van Emden, Contributions to the theory of logic programming, *J. ACM*, **29** (3): 841–862, 1982.
12. G. A. Ringwood, SLD: A folk acronym?, *ACM Sigplan Notices*, **24** (5): 71–75, 1989.
13. L. S. Sterling, E. Y. Shapiro, *The Art of Prolog*, Cambridge, MA: MIT Press, 1986.
14. J. Lobo, J. Minker, A. Rajasekar, *Foundations of Disjunctive Logic Programming*, Cambridge, MA: MIT Press, 1990.

JOHN M. JEFFREY
JORGE LOBO
Elmhurst College
TADAO MURATA
University of Illinois at Chicago