Deductive Databases and Logic Programming

(Winter 2003/2004)

Chapter 4: Built-In Predicates

- Binding Patterns (Modes)
- Range Restriction / Allowedness
- Prolog Built-In Predicates
- Built-In Predicates and Function Symbols

Objectives

After completing this chapter, you should be able to:

- define and explain binding pattern.
- check the allowedness of a clause.
- write Prolog programs using built-in predicates.
- explain how function symbols could be implemented with binding patterns.

Overview

Introduction (1)

- A very pure Prolog Program contains only predicates that are defined by facts and rules.
- However, for larger, real-world applications, this is not very realistic.
- Already for simulating SQL-queries in Prolog, needs e.g. the standard arithmetic operators $+$, $-$, $*$, $/$, and the comparison operators $=$, $\neq$, $<$, $>$, $\leq$, $\geq$.
- For real programs, one needs also a mechanism for input/output etc.

Introduction (2)

- Theoretically, it might be possible to define e.g. $<$ for all numbers that occur in the program by facts.
- But it would at least be tedious to enumerate all facts $X < Y$ that might be important for a program.
- Therefore, Prolog systems and deductive database systems have certain predicates predefined by procedures in the system.
- E.g. for the query $3 < 5$, the system does not look up facts and rules, but calls a built-in procedure written e.g. in C.

Introduction (3)

- Since built-in predicates are defined in the system, it is illegal to write a literal with a built-in predicate in the head of a rule, e.g.

  $X \leq Z \leftarrow X \leq Y \wedge Y \leq Z.$ **Error!**

- Rules contribute to the definition of the predicate in their head, and the definition of built-in predicates cannot be changed.

  Typical error message: "Attempt to modify static procedure $\leq/2$."

- Of course, one can use built-in predicates in the body of a rule (i.e. call them).

Introduction (4)

- Built-in predicates often have restrictions on their arguments: Certain arguments must not be (unbound) variables, but must be known values.

  Whereas in Pure Prolog, predicates have no predefined input and output arguments, now certain arguments can only be input arguments.

- E.g. the query $X > 3$ is not permitted: It would immediately have infinitely many solutions.

  A typical error message is "instantiation fault in $X > 3$".

- But "$p(X) \wedge X > 3$" is permitted: When $X > 3$ is executed, X has already a concrete value.

  If p binds its argument to a value, e.g. "$p(X) \leftarrow$" would not help.

Binding Patterns (1)

Definition:

- A binding pattern for a predicate of arity n (i.e. a predicate with n arguments) is a string over $\{\mathtt{b}, \mathtt{f}\}$ of length n.
- The binding pattern defines which arguments are input arguments, and which are output arguments:
  - ◇ `b`: "bound" (input argument)
  - ◇ `f`: "free" (output argument)
- Binding patterns are not only important for built-in predicates, but can be specified for any predicate.

Binding Patterns (2)

- E.g. consider a predicate sum(X, Y, Z) that is true if and only if X + Y = Z.

  Not every Prolog system has such a predicate, because Prolog uses `is` for evaluating arithmetic expressions, see below.

- The predicate `sum` will typically support the binding pattern `bbf`. This corresponds e.g. to the call sum(3, 5, X).
- It can support also the binding patterns `bfb`, `fbb` (and `bbb`, see below).
- E.g. sum(3, X, 8) binds X to $8 - 3 = 5$.

Binding Patterns (3)

- If all three binding patterns are supported, a deductive DBMS will internally have three procedures:
  - ◇ `sum_bbf(X, Y, var Z): begin Z := X + Y; end`
  - ◇ `sum_bfb(X, var Y, Z): begin Y := Z - X; end`
  - ◇ `sum_fbb(var X, Y, Z): begin X := Z - Y; end`
- The compiler then selects the right procedure depending on the arguments.

  In Prolog, it is not always possible for the compiler to know whether a variable will be bound or free, therefore, there might be a runtime test to check which case applies.

Binding Patterns (4)

- In the example, a predicate plus a binding pattern corresponds to a classical procedure.
- However, in general, a predicate can still have multiple solutions or fail.
- Typically, built-in predicates can succeed only once.

  This is not a strict requirement, but makes the interface simpler. A procedure for a predicate that can fail has a boolean result value.

- Prolog systems and deductive DBMS can usually be extended by adding new built-in predicates written in C or similar languages.

Binding Patterns (5)

Definition:

- A binding pattern $\alpha_1 \ldots \alpha_n$ is more general than a binding pattern $\beta_1 \ldots \beta_n$ iff for all $i \in \{1, \ldots, n\}$: $\alpha_i = \mathtt{b} \implies \beta_i = \mathtt{b}$.

Example/Remark:

- The binding pattern `bbf` is more general than `bbb`.
- One can always use a procedure for a more general binding pattern.
- E.g. the compiler could transform sum(3, 5, 8) into sum(3, 5, X) $\wedge$ X = 8 (with a new variable X).

Binding Patterns (6)

Binding Patterns and Database Relations:

- Database relations have no binding restrictions, i.e. they can be evaluated for the binding pattern `f...f`.

  This is done by a full table scan. E.g. if the relation `father(X,Y)` is stored as a heap file, even the literal `father(arno,Y)` is evaluated like `father(X,Y)` $\wedge$ `X` = `arno`: The system reads every tuple in the relation and checks whether the first attribute has the value `arno`.

- If there is an index on the first attribute, the system uses that index for the binding pattern `bf` (and `bb`). Each index supports a specific binding pattern.

  Since B-tree indexes support also e.g. <-conditions, indexes can more generally be seen as parameterized pre-computed queries.

Binding Patterns (7)

The meaning of "bound":

- There are two different interpretations of what a bound argument is:
  - ◇ Weakly bound: Anything except a variable.
  - ◇ Strongly bound: A ground term.
- Does a complex term with a variable somewhere inside, e.g. "[1,X,2] count as "bound"?
- In deductive databases such "terms with holes" are normally excluded (see "range restriction" below).

  I.e. there is no difference between strongly bound and weakly bound.

Binding Patterns (8)

- In Prolog, complex terms with holes are possible and sometimes useful (e.g. for meta-programming).

  Meta-programming means to process programs as data. This is especially easy in Prolog.

- But then it depends on the predicate where exactly variables might appear.

  E.g. a predicate `length(L,N)` that computes the length `N` of a list `L` could process [1,X,2], but not [1,2|X].

- Thus, it is safest to assume that "bound" means "strongly bound", but that exceptions are possible.

Binding Patterns (9)

- In Prolog, the programmer usually knows the binding pattern for which a predicate is called.

  In contrast to deductive databases, where very different queries must be executed, a Prolog program typically has a "`main`" predicate that calls (directly or indirectly) all other predicates.

- It is common to document this by writing a comment line that lists the predicate with its arguments, where each argument is prefixed with
  - ◇ "+" for bound (input) arguments, and
  - ◇ "-" for free (output) arguments, and
  - ◇ "?" for unrestricted arguments.

Binding Patterns (10)

- E.g.:

```
% length(+L, -N): N is the length of list L.
length([], 0).
length([_|R], N) :- length(R,M), N is M+1.
```

- I.e. the programmer assumes that the first argument is bound and the second argument is free.

  The program might work in other cases, but there is no guarantee. Since the binding pattern appears only in a comment, the Prolog system does not check predicate calls. In some systems, "mode" declarations (that specify binding patterns) help the compiler to optimize the program. Some systems require mode declarations for exported predicates (when modules are separately compiled).

- Ideal logic programs have no binding restrictions!

Semantics of Programs (1)

Formal Treatment of Built-In Predicates:

- Let a fixed interpretation $\mathcal{I}_B$ be given that defines the extensions $\mathcal{I}_B[\![p]\!]$ of all built-in predicates p.

  Of course, this interpretation must also define the domain. Typically, one considers only Herbrand interpretations. Then the base interpretation defines the domain, the meaning of the function symbols, and the meaning of the built-in predicates.

- Then one considers only interpretations $\mathcal{I}$ that are extensions of $\mathcal{I}_B$, i.e. all interpretations must agree with $\mathcal{I}_B$ on the signature for which $\mathcal{I}_B$ is defined.

Semantics of Programs (2)

- One identifies an interpretation $\mathcal{I}$ with the set of facts $p(t_1, \ldots, t_n)$ with $\mathcal{I} \models p(t_1, \ldots, t_n)$, where p is not a built-in predicate.

  This is an extension of the corresponding convention for Herbrand interpretations. Since built-in predicates nearly always have an infinite extension, excluding them increases the chances that the set of facts is finite (and thus can be explicitly written down or explicitly stored). Of course, also predicates defined by rules can have infinite extensions.

- As before, the semantics of a program P is the least fixed point of the T_P-operator (see next slide).

  This is the least model of P among the interpretations $\mathcal{I}$ that extend $\mathcal{I}_B$.

Semantics of Programs (3)

- The definition of the immediate consequence operator T_P does not have to be modified:

  $T_P(\mathcal{I}) := \{F \in \mathcal{B}_\Sigma \mid$ There is a rule
  $A \leftarrow B_1 \wedge \cdots \wedge B_n$ in P
  and a ground substitution θ,
  such that
  - $\mathcal{I} \models B_i\theta$ for $i = 1, \ldots, n$, and
  - $F = A\theta\}$.

- Since the head literal A does not contain a built-in predicate, one gets only facts about user-defined predicates.

Valid Binding Patterns (1)

- The designer of the logic programming system defines for each built-in predicate p a set $valid_B(p)$ of valid binding patterns such that for all built-in predicates p and all $\beta \in valid_B(p)$:
  - ◇ Suppose that p has arity n and let
    $$\{i_1, \ldots, i_k\} := \{i \mid 1 \leq i \leq n,\ \beta_i = \mathtt{b}\}.$$
  - ◇ Then for all domain elements $t_{i_1}, \ldots, t_{i_k}$ in $\mathcal{I}_B$, the set
    $$\{(t'_1, \ldots, t'_n) \in \mathcal{I}_B[\![p]\!] \mid t'_{i_1} = t_{i_1}, \ldots, t'_{i_k} = t_{i_k}\}$$
    must be finite and computable.

Valid Binding Patterns (2)

- I.e. the requirement is that given any values for the bound arguments, it must be effectively possible to compute values for the other arguments, and to compute all such solutions.
- Together with the range-restriction defined below, this ensures that each single application of the T_P-operator is computable and has a finite result.

  Of course, in the limit (minimal model), it is still possible that user-defined predicates have infinite extensions. Which is also bad, because it means that the computation does not terminate.

Overview

1. Built-In Predicates, Binding Patterns
2. Important Built-In Predicates in Prolog
3. Range-Restriction, Allowedness
4. Function Symbols and Built-In Predicates

Term Comparison (1)

- t_1 `=` t_2: t_1 and t_2 are unifiable.
- t_1 `==` t_2: t_1 and t_2 are textually identical.

  E.g. `X = a` is true and has the side effect of binding `X` to `a`. However, `X == a` is false (unless `X` was already bound to `a`).
- t_1 `\=` t_2: t_1 and t_2 are not unifiable.
- t_1 `\==` t_2: t_1 and t_2 are not textually identical.
- t_1 `@<` t_2: t_1 is before t_2 in the standard term order.

  The standard order of terms is explained on the next slide.
- t_1 `@>` t_2: t_1 is after t_2 in the standard term order.

Term Comparison (2)

- t_1 `@=<` t_2: Equivalent to t_1 `@<` t_2 or t_1 `==` t_2.
- t_1 `@>=` t_2: Equivalent to t_1 `@>` t_2 or t_1 `==` t_2.
- The standard order of terms is partially system dependend (despite its name), but often one needs only any order.

  E.g. first variables (in undefined sequence), then atoms (alphabetically), then strings (alphabetically), then numbers (in the usual order), then compound terms (first by arity, then by name, then recursively by arguments from left to right).

- `compare`(o,t_1,t_2): Binds o to <, =, or >.

Term Classification (1)

- Terms have different types, e.g. integers, atoms, variables. There are various type test predicates:
  - ◇ `var`(t): t is an unbound (free) variable.
  - ◇ `nonvar`(t): t is not an unbound variable.
  - ◇ `atom`(t): t is an atom, e.g. `abc`.
  - ◇ `atomic`(t): t is an atom or a number.

    Depending on the system, also strings might count as atomic.

  - ◇ `integer`(t): t is an integer.
  - ◇ `float`(t): t is a floating-point number.

    Depending on the system, this might also be called `real`(t).

Term Classification (2)

- Type test predicates, continued:
  - ◇ `number`(t): t is integer or floating-point number.
  - ◇ `string`(t): t is a string.

    This exists only in systems that represent strings as a data type of its own, not as lists of ASCII codes.

  - ◇ `compound`(t): t is a compound term, e.g. `f(X)`.
  - ◇ `callable`(t): t is atom or compound term.
- The result depends on the current execution state, e.g. `var(X), X=2` succeeds, but `X=2, var(X)` fails.

  If one uses such predicates, one cannot rely on the commutativity of conjunction.

Term Manipulation

- `functor`(t,f,n): t is a term with functor f, arity n.

  This can be used either to extract the functor from a term (binding pattern `bff`) or to construct a term with the given functor and n distinct variables as arguments (binding pattern `fbb`).

- `arg`(n,t,a): a is the n-th argument of t.
- t `=..` L: L is a list consisting of the functor and the arguments of t. E.g. `f(a,b) =.. [f,a,b]`.

  This predicate is called "univ". It can be used in both directions (binding pattern `bf` and `fb`).

- `name`(A,L): Atom A consists of characters `L`.

  E.g. `name(abc,[97,98,99])`. Binding patterns: `bf` and `fb`.

Arithmetic Predicates (1)

- Arithmetic expressions can be evaluted with the built-in predicate `is`, e.g. `X is Y+1`.
- `is` is defined as infix operator (`xfx`) of priority 700.
- The arithmetic expression can contain `+`, `-` (uniary and binary), `*`, `/`, `//` (integer division), `mod` (modulo), `/\` (bit-and), `\/` (bit-or), `<<` (left shift), `>>` (right shift), `\` (bit complement).

  Possibly also functions such as `sin`, `cos`, etc. can be used.
- The right argument must be variable-free, i.e. `is` supports only the binding patterns `fb` and `bb`.

Arithmetic Predicates (2)

- Arithmetic comparison operators first evaluate expressions on both sides before they do the comparison. E.g. `X + 1 < Y * 2` is possible.
- Both arguments must be variable-free (binding pattern `bb`). Of course, bound variables are no problem.
- Arithmetic comparison operators are:
  `=:=` ($=$), `=\=` ($\neq$), `<` ($<$), `>` ($>$), `=<` ($\leq$), `>=` ($\geq$).

  The equality test is written `=:=`, because `=` is already the unification (which does not evaluate arithmetic expressions). In the same way, inequality is written `=\=`, because `\=` means "does not unify with". Note that $\leq$ is written `=<`, because the Prolog designers wanted to save the arrow `<=` for other purposes.

String Functions (1)

- SWI-Prolog has strings as new datatype.

  As explained above, classical Prolog systems represent strings as lst of characters. ECLiPSe Prolog also has a string data type, but the functions have different names.
- Conversion functions:
  - ◇ `string_to_atom`(s, a): Conversion between string and atom.

    Despite its name, both directions are supported (binding patterns `bf` and `fb`).
  - ◇ `string_to_list`(s, L): Conversion between string and list of ASCII codes.

String Functions (2)

- Other string functions:
  - ◇ `string_length`(s, n): Computes the number of characters in s.
  - ◇ `string_concat`(s_1, s_2, s_3): String concatenation.

    Supports binding patterns `ffb`, `bbf`. There is also `atom_concat`.
  - ◇ `sub_string`(s_1,n_1,n_2,n_3,s_2): s_2 is substring of s_1.

    The substring starts at position n_1 (i.e. there are n_1 characters before the match), it has length n_2, and there are n_3 characters (in s_1) after the match. There is also `sub_atom`.

Exercises

- Define a predicate to compute the Fibonacci numbers:
$$f(n) := \begin{cases} 1 & n = 0,\ n = 1 \\ f(n-1) + f(n-2) & n \geq 2. \end{cases}$$
- Define $\texttt{sum}(\texttt{X}, \texttt{Y}, \texttt{Z})$ that holds iff $\texttt{X} + \texttt{Y} = \texttt{Z}$ and can handle the binding patterns `bbf`, `bfb`, `fbb`, `bbb`.
- Define a predicate $\texttt{makeground}(t)$ that binds all variables that appear in t to `x`.

Constructed Goals

- Proof goals can be dynamically constructed, i.e. can be computed at runtime.

  In purely compiled languages, that is not possible.
- $\texttt{call}(A)$: Executes the literal A.
- In many Prolog systems, one can write simply `X` instead of `call(X)`.

  But it might be clearer to use `call`.

All Solutions (1)

- $\texttt{findall}(X,\ A,\ L)$: L is the list of all X such that A is true.
- E.g. given the facts $p(a)$ and $p(b)$,

  `findall(X, p(X), L)`

  returns `L = [a, b]`.
- It is not required that the first argument is a variable, it could also be e.g. `f(X,Y)` if one is interested in bindings for both variables.

  I.e. in general, the result list contains the instantiation of the first argument whenever a solution to the second argument was found.

All Solutions (2)

- $\texttt{bagof}(X,\ A,\ L)$: (similar to `findall`).

  The different lies in the treatment of variables that occur in A, but do not occur in X. `findall` treats them as existentially quantified, i.e. it does not bind them, and `findall` can succeed only once. In contrast, `bagof` binds such variables to a value and collects then only solutions with this value. Upon backtracking, one can also get other solutions. For example, suppose that `p` is defined by the facts `p(a,1)`, `p(a,2)`, `p(b,3)`. Then `findall(X, p(Y,X), L)` would bind `L=[1,2,3]`. However, `bagof(X, p(Y,X), L)` would succeed two times: One for `Y=a` and `L=[1,2]`, and once for `Y=b` and `L=[3]`.
- $\texttt{setof}(X,\ A,\ L)$: As `bagof`, but the result list is ordered and does not contain duplicates.

Dynamic Database (1)

- Prolog systems permit that the definition of certain predicates is modified at runtime.
- E.g. if a database relation is represented as a set of facts, one can insert and delete facts.
- Such changes persist even when Prolog backtracks to find another solution.

  Input/output and modifications of the dynamic database are the only changes that are not undone upon backtracking.

Dynamic Database (2)

- Since modern Prolog systems normally compile predicates, one must explicitly declare predicates that can be modified at runtime:

  `:- dynamic(`p/n`).`

- `assert(`φ`)`: The clause φ is inserted into the dynamic database.

  Normally, φ will be a fact, but it is also possible to assert rules. Some Prolog systems guarantee that the new clause is appended at the end of the predicate definition, but officially, there is no guarantee about the order unless one uses `asserta` (insert at the beginning) or `assertz` (insert at the end).

Dynamic Database (3)

- `retract(`φ`)`: Remove a clause from the database.
- `retractall(`A`)`: All rules for which the head unifies with A are removed from the database.

  In ECLiPSe it is `retract_all`. The call `retractall(`A`)` succeeds also when there are no facts/rules that match A.

- `abolish(`p`, `n`)`: Remove the definition of p/n.

  Then the predicate is no longer defined at all. A call to the predicate would give an error.

- `listing`: Lists the dynamic database.

Dynamic Database (4)

Exercise:

- Define a predicate `next(N)`, that generates unique numbers, i.e. the first call returns 1, the second call returns 2, and so on.
- Define a predicate `all_solutions` that works like `findall`.

  Of course, you should not use `findall` or `bagof`, but the dynamic database. For simplicity, you can assume that the goal does not call recursively `all_solutions`. You need the predicate `fail` that is logically false (triggers backtracking).

Input/Output (1)

Input/Output of Terms:

- `write(`t`)`: Print term t (using operators).

  E.g. `write(1+1)` and `write(+(1,1))` both print `1+1`. The predicate `write` does not know how the term was originally written, it only gets the internal data structure as input. Normally also variable names are lost when the term is represented internally, therefore output variable names might appear strange (e.g. `write(X)` might print `_G219`, where `219` is probably a memory address).

- `display(`t`)`: Print term t in standard syntax.

  E.g. `display(1+1)` prints `+(1, 1)`.

- `writeq(`t`)`: Print t, put atoms in `'...'` (if necessary).

  This guarantees that the term can be read again with `read`, see below.

Input/Output (2)

Input/Output of Terms, continued:

- `write_canonical(`t`)`: Print t in standard syntax, put atoms in `'...'` (if necessary).

  This is even safer than `writeq` for reading the term again, because the current operator declarations are not needed. New Prologs have a predicate `write_term(`t`,`O`)` that prints t with options O. Then `write`, `display`, etc. are abbreviations for `write_term` with certain options.

- `nl`: Print a line break.

- `read(`X`)`: Read a term, bind X to the result.

  The input term must be terminated with "`.`⟨Newline⟩". At the end of the file, most Prolog systems return $X =$ `end_of_file`. Together with operator declarations, `read` is already a quite powerful parser.

Input/Output (3)

Input/Output of Characters:

- `put_code(`C`)`: Print character with ASCII-code C.

  In older Prolog versions (compatible to DEC-10 Prolog), this is simply called `put`. The newer `put_code` is contained in the ISO Standard.

- `get_code(`C`)`: Read next character, unify C with its ASCII-code.

  At the end of file, C is set to -1. In older Prolog versions, this is called `get0`. The predicate `get` first skipped spaces, and then unified C with the next non-space character.

- `peek_code(`C`)`: Unify C with ASCII-code of next input character without actually reading it.

Input/Output (4)

	0	1	2	3	4	5	6	7	8	9
0	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	␣	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL		

Input/Output (5)

Input/Output of Characters, continued:

- It is also possible to work with one-character atoms instead of ASCII-codes:
  - `put_char`(C): Print atom as character.

    E.g. to print a space one writes `put_char(' ')`.
  - `get_char`(C): Read next character, unify C with corresponding atom.

    E.g. if the user enters "a", `get_code(C)` returns `C=97`, whereas `get_char(C)` returns `C=a`. At the end of file, C is set to `end_of_file`.
  - `peek_char`(C): Unify C next input character (as atom) without actually reading it.

Input/Output (6)

Input/Output of Characters, continued:

- The above `get_*`-predicates normally wait for an entire line of input from the keyboard.
- In contrast to `read`, it is not necessary to finish the input with ".", `Enter/Return` suffices.
- Every Prolog system has a way to read characters without buffering, but that is system dependent.

  E.g. in SWI Prolog, use `get_single_char`. In GNU Prolog, use `get_key`.

Input/Output (7)

File Input/Output:

- In Prolog, open files are called "streams".
- Streams are actually a bit more general:
  - Of course, also keyboard input and screen output are streams (called `user_input` and `user_output`).
  - Some Prolog systems support also pipes (for inter-process communication), sockets (for network communication), and I/O from atom names and ASCII-code lists/strings.

Input/Output (8)

File Input/Output, continued:

- All of the above I/O predicates have also a version with an additional argument for a stream.
- E.g. `write`(S, t) prints term t to stream S.
- `open`(F, M, S): Opens file F in mode M (`read`, `write`, `append`, possibly `update`), and returns stream S.

  The file name F should be an atom. At least SWI-Prolog also supports `pipe`(C) with a command C. There is also `open`(F,M,S,O) that has in addition a list O of options, e.g. `[type(binary)]`. One can also use the option `[alias`(A)`]` to declare atom A as stream name which can be used in calls to `write` etc. (instead of the stream object S itself).

Input/Output (9)

File Input/Output, continued:

- Files open in binary mode must be read or written with `put_byte/get_byte/peek_byte` instead of the `put_code/get_code/peek_code` predicates.

  The difference is that the Prolog system might do operating system dependent translations for text files, e.g. map CR/LF to LF under Windows, whereas binary files are read verbatim. At least GNU Prolog produces a runtime error (exception) if one uses `get_byte` on a text file or vice versa.

- `close(`S`)`: Close stream S.

  There is also `close(`S, O`)` with options O. If an output file is not closed, the buffer might not be written, and data is lost.

Input/Output (10)

File Input/Output, continued:

- `flush_output/flush_output(`S`)`:
  Flush pending (buffered) output.
- `at_end_of_stream/at_end_of_stream(`S`)`:
  Succeeds after the last character was read.
- `set_input(`S`)`: Set the default input stream to S.
- `set_output(`S`)`: Set the default output stream to S.
- `get_input(`S`)`: Set S to the current input stream.
- `get_output(`S`)`: Set S to the current output stream.

Input/Output (11)

Exercises:

- DEC-10 Prolog had a predicate `tab(`N`)` that printed N spaces. Please define it.

  Since `tab` is still contained in some Prologs, it might be necssary to use a different name, e.g. `nspaces`.

- Define a predicate `calc` that prints a prompt, reads an arithmetic expression (without variables), evaluates it, prints the result, and so on until the user enters "`quit`".

  You can assume that the user ends each input line with "`.`". Furthermore, you do not have to handle syntax errors.

Control (1)

- A`,` B: A and B (conjunction).
- A`;` B: A or B (disjunction).

  Conjunction binds stronger than disjunction (";" has priority 1100, "," has priority 1000). One can use parentheses if necessary. Disjunction is not strictly needed, one can use several rules instead.

- `true`: True (always succeeds).
- `fail`: False (always fails).

  Obviously, this can only be interesting with previous side effects (or the cut). Examples are shown in the next chapter.

- `repeat`: Always succeeds, also on backtracking.

  This can be defined as `repeat.` `repeat :- repeat.`

Control (2)

- `!`: Ignore all previous alternatives in this predicate activation (cut, see next Chapter).

  This means that no further rules for the same predicate will be tried, and no further solutions for all body literals to the left of the cut.

- A `->` B_1`;` B_2: If A, then B_1, else B_2.

  This really means (A `->` B_1)`;` B_2. The arrow "->" has priority 1050, disjunction ";" has priority 1100.

- A `->` B: If A, then B, else fail.

  This is equivalent to A`,` `!``,` B.

- `once(`A`)`: Compute only first solution for A.

Control (3)

- `\+` A: A is not provable (fails).

  This is called negation as failure. It is not the logical negation, because Prolog permits only to write down positive knowledge. Negation as failure behaves non-monotonically, whereas classical predicate logic is monotonic: If one adds formulas, one can at least prove everything that was provable earlier. Some Prologs also understand `not` A.

Prolog Environment (1)

- `halt`: Leave the Prolog system.
- `abort`: Stop the current Prolog program.

  Control returns to the top-level Prolog prompt. This predicate is not contained in the ISO standard.

- `help(`p/n`)`: Show online manual for predicate p of arity n (not in all Prolog systems).
- `shell(`C`,`E`)`: Execute the operating system command C, unify E with the exit status.

  This is not contained in the ISO standard. If the predicate is missing, look for `system/1`, `unix/1`, and `shell/1`. There might be many more predicates to give Prolog an operating system interface.

Prolog Environment (2)

- `statistics`: Display statistics, such as used CPU time, used size of various memory areas, etc.

  This predicate is not contained in the ISO Prolog standard. There might also be `statistics/2` to query specific statistics.

- `trace`: Switch debugger on (in creep mode).

  Creep mode means step by step execution.

- `spy(`p/n`)`: Set a breakpoint on predicate p of arity n.
- `debug`: Switch debugger on (in leap mode).

  Leap mode means that execution stops only at breakpoints.

- `notrace/nodebug`: Switch debugger off.

Predicate Documentation (1)

Meaning of the Predicate:

- Purpose/Function of the predicate ("synopsis").

  If the predicate name is an abbreviation, what is the full version? Use meaningful names for the arguments.

- Reasons for the truth value false ("fails").

  It is best to specify which mathematical relation is defined by the predicate.

- Behaviour on backtracking ("resatisfiable?").

  Can there be several solutions?

- Side effects.

  Input/output, changes of the data base, changes in system settings.

Predicate Documentation (2)

Reasons for Error Messages (Exceptions):

- Type-restrictions for arguments.

  E.g. it must be a number, a callable term, etc.

  Which arguments must be free/bound?

  Usually, arguments that must be bound are prefixed with "+", and arguments that should be unbound variables are marked with "–". "?" marks an argument without restrictions.

Further Information:

- Examples.
- Related predicates ("see also").

Overview

1. Built-In Predicates, Binding Patterns
2. Important Built-In Predicates in Prolog
3. Range-Restriction, Allowedness
4. Function Symbols and Built-In Predicates

Motivation (1)

- In deductive databases, query evaluation is done by applying the T_P-operator iteratively to compute the minimal model (with certain optimizations).

  Computing the entire minimal model would not be goal-directed. Of course, one should compute only facts that are important for the query. This problem is solved by the magic set transformation: Given a logic program and a query, it computes a new logic program that has the same answer, but implies only facts relevant to the query. See Chapter 6.

- The allowed rules must be restricted so that immediate consequences can effectively be computed.

Motivation (2)

- For example, computing immediate consequences for a rule like the following would be difficult:

$$p(X,Y) \leftarrow q(X).$$

- The possible values for Y depend on the domain: All data values can be inserted, often this set infinite, and maybe not even explicitly known.

  Normally, one works with the Herbrand universe that consists of all terms which can be constructed from the constants and function symbols appearing in the program. Then one can add a completely unrelated fact, in which a new constant appears, and thereby change the extension of p. That is a strange behaviour.

Motivation (3)

- Given e.g. $q(a)$, one could derive the “fact” $p(a,Y)$.
- One problem with this is that variables cannot be easily represented in database relations.

  As long as one does not use function symbols, derived predicates should correspond to views in relational databases. Furthermore, at least some prototypes did actually use a relational database system for query evaluation: Then storing an intermediate result in a temporary relation for p is at least difficult.

- In contrast to Prolog, deductive databases normally use only one-directional, restricted form of unification (“matching”): Variables appear only in rules, body literals are matched with variable-free facts.

Allowed Rules (1)

Definition (Allowed Rule, First Try):

- A rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ is called allowed iff every variable that appears in the head literal A appears also in at least one body literal B_i.

Note:

- This definition works only if the body literals have no binding restrictions.
- E.g. one cannot compute all consequences of the following rule, although it statisfies the condition:

```
less(X, Y) :- X < Y.
```


Allowed Rules (2)

Definition (Allowed Rule with Built-In Predicates):

- A rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ is called allowed iff every variable that appears in the rule appears also in at least one body literal B_i, the predicate of which is not a built-in predicate.

Example:

- E.g. the following rule satisfies this condition:

```
teenager(X) :- person(X, BirthYear),
               BirthYear < 1995,
               BirthYear > 1984.
```

Allowed Rules (3)

Note:

- If all rules are allowed in the above sense, one can effectively compute every approximation $T_P \uparrow i$ of the minimal model.

  Assuming that $valid_B(p)$ is not empty for every built-in predicate p. Built-in predicates without any valid (implemented) binding pattern obviously make no sense.

- If in addition, the rules do not contain function symbols, the minimal model itself can be computed.

  It contains then only constants that appear in the program. Assuming that programs are always finite, this means that the minimal model is finite, and is reached after finitely many iterations of the T_P-operator.

Range Restriction (1)

- The notion of "allowed rule" above assumes that we really want to compute the entire extension of all derived predicates.

  I.e. that all predicates should support the binding pattern `f...f` like stored relations. This is not always required.

- Predicates that have binding restrictions are typically defined by rules that are not allowed:

```
append([], L, L).
append([F|R], L, [F|RL]) :-
      append(R, L, RL).
```


Range Restriction (2)

- The rule about `less` makes sense if it is called with binding pattern `bb`:

```
less(X, Y) :- X < Y.
```

- Occurrences of variables in literals with built-in predicates can act as binding. This rule defines a predicate without any binding restriction:

```
price_with_vat(Prod, X) :- product(Prod, Price),
                           X is Price * 1.16.
```

- All this shows that the allowedness requirement is too restrictive.

Range Restriction (3)

Definition (Input Variables):

- Given a literal $A = p(t_1, \ldots, t_n)$ and a binding pattern $\beta = \beta_1, \ldots, \beta_n$ for p, the set of input variables of A with respect to β is

  $$input(A, \beta) := \bigcup \{vars(t_i) \mid 1 \le i \le n,\ \beta_i = \mathtt{b}\}$$

  (i.e. all variables that appear in bound arguments).

Note:

- Input variables in body literals must be bound before the literal can be called. Input variables in head literals are bound when the rule is executed.

Range Restriction (4)

Definition (Valid Binding Patterns):

- Let $valid$ be a function that maps every predicate p to a set $valid(p)$ of binding patterns for p.
- For built-in predicates p: $valid(p) = valid_B(p)$.
- $valid$ is called a valid binding pattern specification.

Remark:

- We assume that the programmer defines valid binding patterns for every predicate.

  In practice, it might be possible to compute the possible binding patterns, but that would complicate the next definition.

Range Restriction (5)

Definition (Range-Restricted Rule):

- A rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ is range-restricted for a binding-pattern β iff there is a sequence $i_1, \ldots, i_n$ of the body literals (i.e. $\{i_1, \ldots, i_n\} = \{1, \ldots, n\}$) such that
  - ◇ for every $j \in \{1, \ldots, n\}$ there is a binding pattern $\beta_j \in valid(pred(B_{i_j}))$ with $input(B_{i_j}, \beta_j) \subseteq input(A, \beta) \cup vars(B_{i_1} \wedge \cdots \wedge B_{i_{j-1}})$
  - ◇ and furthermore it holds that
    $$vars(A) \subseteq vars(B_1, \ldots, B_n) \cup input(A, \beta).$$

Range Restriction (6)

- The definition assumes the following evaluation:
  - ◇ First, variables in the bound argument positions of the head literal are assigned values (based on the input arguments of the predicate call).
  - ◇ Then the body literals are evaluated in some order. For each literal, variables in bound argument positions must already have a value. Variables in other argument positions get a value by this call.
  - ◇ In the end, a tuple is produced that corresponds to the head literal. Therefore, all variables in the head literal must now have a value.

Range Restriction (7)

- The evaluation sequence of body literals may depend on the binding pattern for the head literal.
- For instance, consider the following rule:

```
p(X,Y) :- sum(X,1,Z), prod(Z,2,Y).
```

- The given sequence of body literals is possible for the binding pattern `bf`.
- For the binding pattern `fb`, the system should automatically switch the sequence of body literals.

  The Datalog programmer does not necessarily know the binding pattern. Furthermore, it would be bad style to double the rule.

Range Restriction (8)

Definition (Range-Restricted Program):

- A program P is range-restricted with respect to a binding pattern specification $valid$ iff for every rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ and every binding pattern $\beta \in valid\big(pred(A)\big)$, the rule is range-restricted for β.
- A program P is strictly range-restricted iff every rule in P is range-restricted for the binding pattern `f...f`.

  Strict range restriction is the requirement for the T_P-operator to be directly executable. As we will see, the magic set transformation turns a range-restricted program into a strictly range-restricted program.

Overview

Record Constructors (1)

- Function symbols in Prolog are record/structure constructors.
- Let `cons`(E, N, L) be a built-in predicate for managing nodes L in a linked list (records with two components: list element E and "next" pointer N).
- It can be called with two binding patterns:
  - ◇ `bbf`: For constructing a list node.
  - ◇ `ffb`: For selecting the components of a list node.
- Note that `cons`(E, N, L) actually means L=`[`E`|`N`]`.

Record Constructors (2)

- Consider again the definition of `append`:

```
append([], L, L).
append([F|R], L, [F|RL]) :-
        append(R, L, RL).
```

- Instead of using composed terms like `[F|R]`, one can also use the built-in predicate `cons`:

```
append([], L, L).
append(L1, L2, L3) :-
        cons(F, R, L1),    % Split L1 into F and R
        append(R, L2, RL),
        cons(F, RL, L3).   % Compose F and RL to L3
```

Record Constructors (3)

- The right sequence of body literals in the above rule depends on the binding pattern for the predicate.
- As written down, the rule works if `append` is called with binding pattern `bbf`.
- If it is called e.g. with binding pattern `ffb`, the body literals should be (automatically) reordered:

```
append(L1, L2, L3) :-
        cons(F, RL, L3),
        append(R, L2, RL),
        cons(F, R, L1).
```


Record Constructors (4)

- If for every function symbol f of arity n, one has a built-in predicate p_f of arity $n+1$ that constructs/splits records of type f, composed terms are not strictly necessary:
  - ◇ As shown above for `append`, one can always replace them by a new variable and a call to the built-in predicate.
- Of course, this assumes that there are no terms with "holes" (variables) in them.

  In deductive databases, this is normally the case.

Evaluable Functions (1)

- Conversely, one could use functional notation for certain built-in predicates.
- E.g. consider

```
fib(0, 1).
fib(1, 1).
fib(N, F) :-
        N > 1,
        N1 is N-1, N2 is N-2,
        fib(N1, F1), fib(N2, F2),
        F is F1+F2.
```


Evaluable Functions (2)

- One could now write the rule as (not correct in Prolog, typical beginner's error!):

```
fib(N, F) :-
        N > 1,
        fib(N-1, F1), fib(N-2, F2),
        F is F1+F2.
```

- Even the following would be possible:

```
fib(N, fib(N-1)+fib(N-2)) :- N > 1.
```

- A preprocessor could translate both back to the standard predicate notation.

Evaluable Functions (3)

- So, why has Prolog only non-evaluable functions (record constructors)?
- Record constructors are uniquely invertable.
- Consider e.g. the following rule:

```
p(X+Y, X, Y).
```

  Compare it with this rule:

```
q([X|Y], X, Y).
```

- The first rule does not support the binding pattern `bff`, the second does support it.

Evaluable Functions (4)

- As long as one has only record constructors, every occurrence of a variable in a bound argument position in the head defines a value for the variable.
- For evaluable functions, this is not necessarily the case. But e.g. the following rule supports `bf`:

```
p(X+1, X).
```

- However, new logic programming languages are still being proposed, and everybody is free to define (and implement) his/her own language.

  There are many proposals for combined logic-functional languages.