Deductive Databases and Logic Programming

(Summer 2011)

Chapter 5: Practical Prolog Programming

- The Cut and Related Constructs
- Prolog vs. Pascal
- Definite Clause Grammars

Objectives

After completing this chapter, you should be able to:

- explain the effect of the cut.
- write Prolog programs for practical applications.
- use context-free grammars in Prolog.

Overview

1. The Cut and Related Constructs

2. Prolog vs. Pascal

3. Definite Clause Grammars

The Cut: Effect (1)

- The cut, written "`!`" in Prolog, removes alternatives that otherwise would have been tried during backtracking. E.g. consider this rule:

  $p(t_1,\ldots,t_k)$ `:-` A_1, ..., A_m, `!`, B_1, ..., B_n.

- Until the cut is executed, processing is as usual.
- When the cut is reached, all previous alternatives for this call to the predicate p are removed:
  - ◇ No other rule about p will be tried.
  - ◇ No other solutions to the literals A, ..., A_m will be considered.

The Cut: Effect (2)

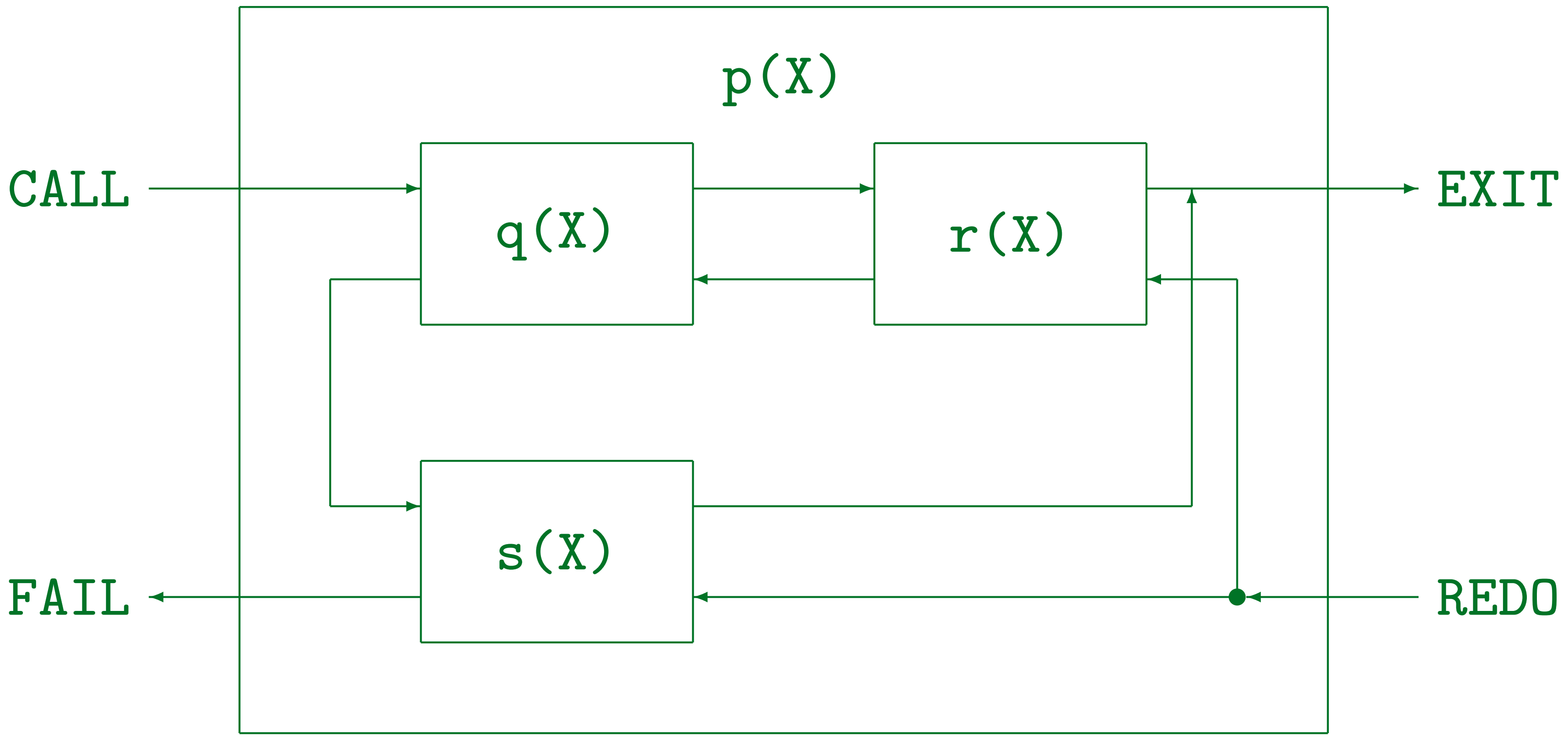
- Example:

```
p(X) :- q(X), !, r(X).
p(X) :- s(X).
q(a).
q(b).
r(X).
s(c).
```

- With the cut, the query `p(X)` returns only `X=a`.
- Without the cut, the solutions are `X=a`, `X=b`, `X=c`.
- Exercise: Can the second rule about `p` ever be used?

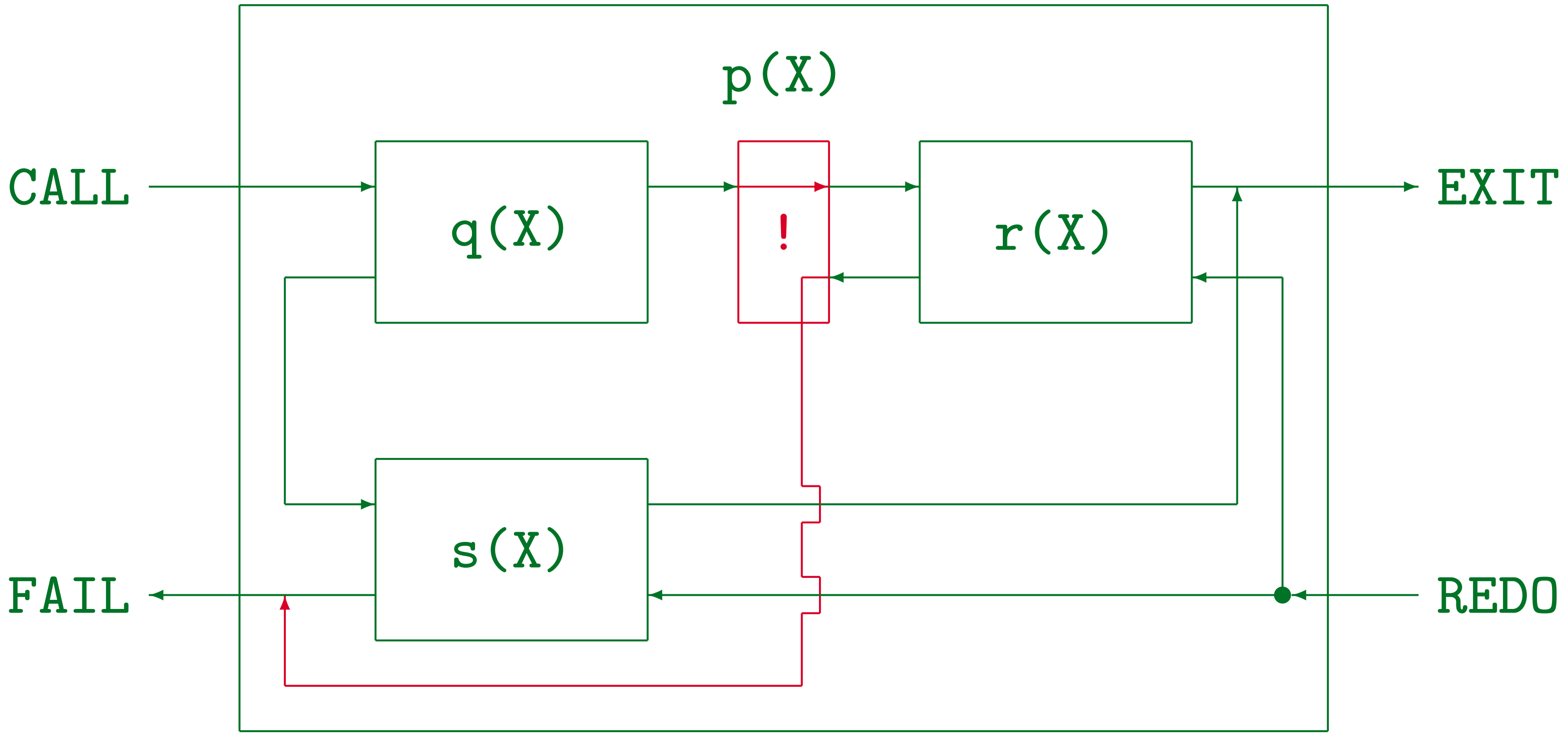
The Cut: Effect (3)

Four-Port Model without Cut:



The Cut: Effect (4)

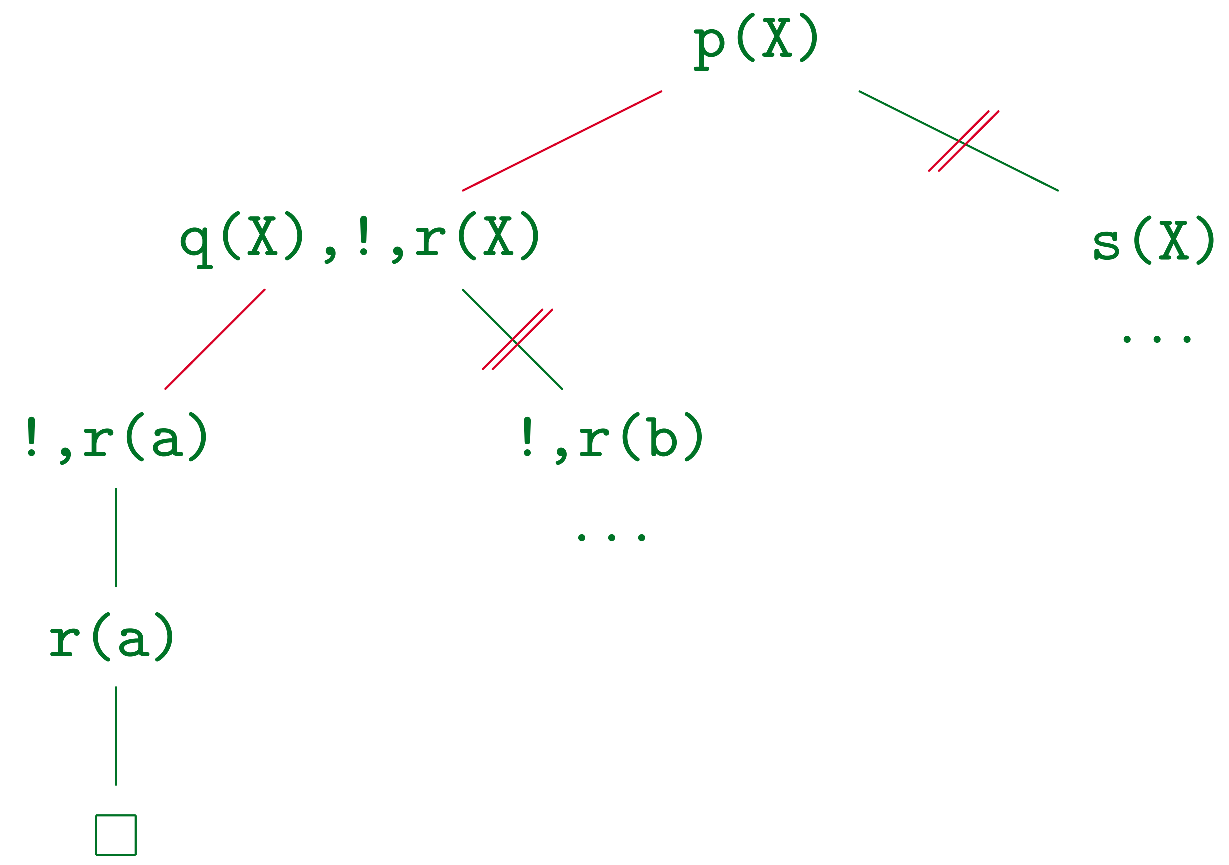
Four-Port Model with Cut:



The Cut: Effect (5)

- A call to the cut immediately succeeds (like `true`).
- Any try to redo the cut not only fails, but immediately fails the entire predicate call.
- In the SLD-tree, the cut "cuts away" all still open branches between
  - ◇ the node where the cut was introduced (i.e. the child of which contains the cut), and
  - ◇ the node where the cut is the selected literal.

The Cut: Effect (6)



The Cut: Effect (7)

- Before and after the cut, the backtracking is normal (only not through the cut):

```
p(X,Y) :- q1(X), q2(X,Y).
p(X,Y) :- r1(X), r2(X), !, r3(X,Y), r4(Y).
p(X,Y) :- s(X,Y).
q1(a).
q1(b).     q2(b,c).
r1(d).
r1(e).     r2(e).
r3(e,f).
r3(e,g).   r4(g).
s(h,i).
```

- The query `p(X,Y)` has solutions `X=b,Y=c` and `X=e,Y=g`.

Cut: Improving Runtime (1)

- One application of the cut is to improve the runtime of a program by eliminating parts of the proof tree that cannot yield solutions or at least cannot yield any new solutions.
- Consider the predicate `abs` that computes the absolute value of a number:

```
abs(X,X) :- X >= 0.
abs(X,Y) :- X =< 0, Y is -X.
```

- When the first rule is successful, it is clear that the second rule does not have to be tried.

Cut: Improving Runtime (2)

- Consider now the goal `p(X), abs(X,Y), Y > 5` with the facts `p(3)`, `p(0)`, `p(-7)`.
- First `p(X)` succeeds with `X` bound to 3, then `abs(3,Y)` succeeds for `Y=3`, but then `3 > 5` fails.
- Now backtracking would normally first try to find an alternative solution for `abs(3,Y)`, since there is another rule about `abs` that has not yet been tried.
- This is obviously useless, and the runtime can be improved by immediately backtracking to `p(X)`.

Cut: Improving Runtime (3)

- With the cut, one can tell the Prolog system that when the first rule succeeds, the second rule cannot give anything new:

```
abs(X,X) :- X >= 0, !.
abs(X,Y) :- X =< 0, Y is -X.
```

- Of course, one could have (should have) written the condition in the second rule `X < 0`.
- Then some (but not all) Prolog systems are able to discover themselves that the rules are mutually exclusive.

Cut: Improving Space (1)

- Making clear that a predicate has no other solution improves also the space (memory) efficiency.
- The Prolog system must keep a record ("choice-point") for each predicate call that is not yet complete (for backtracking into the predicate call later).
- Even worse, certain data structures within the Prolog system must be "frozen" when it is necessary to support later backtracking to this state.
- Then e.g. variable bindings must be logged (on the "trail") so that they can later be undone.

Cut: Improving Space (2)

- In imperative languages, when a procedure call returns, its stack frame (containing local variables and other information) can be reused.
- In Prolog, this is not always the case, because it might be necessary to reactivate the procedure call and search for another solution.
- E.g. consider the following program:

```
p(X) :- q(X), r(X).
q(X) :- s(X), t(X).
s(a). s(b). t(a). t(b). r(b).
```

Cut: Improving Space (3)

- The call `q(X)` first exits with `X=a`, but then `r(a)` fails, thus the call `q(X)` is entered again, which in turn reactives `s(X)`.

  Upon backtracking, also the binding of `X` must be undone.

- In the above example, not much can be improved, because there really are alternative solutions.
- However, when a predicate call has only one solution, it should be executed like a procedure call in an imperative language.

Cut: Improving Space (4)

- Predicate calls that can have at most one solution are called deterministic.

  Sometimes one calls the predicate itself deterministic, but then one usually has a specific binding pattern in mind. E.g. `append` is deterministic for the binding pattern `bbf`, but it is not deterministic for `ffb`.

- For efficient execution, it is important that the Prolog system understands that a predicate call is deterministic. Here a cut can help.

  Actually, the cut in the definition of `abs` makes the predicate deterministic. In general, it might be important that `abs(0,X)` succeeds "two times", Prolog is not allowed to automatically remove one solution. Deductive databases are set-oriented, there more powerful optimizers are possible.

Cut: Improving Space (5)

- Consider `abs` applied to a list:

```
abs_list([], []).
abs_list([X|R], [Y|S]) :- abs(X, Y),
                          abs_list(R, S).
```

- When the Prolog system thinks that `abs` is nondeterministic, it will keep the stackframe for each call to `abs` (and for the calls to `abs_list`).
- When a predicate calls a nondeterministic predicate, it automatically becomes nondeterministic, too.

  Only for the last body literal of the last rule about a predicate, the stack frame of the predicate is reused (under certain conditions), and thus does not remain, even when this body literal is non-deterministic.

Cut: Improving Space (6)

- In the above example, making `abs` deterministic (by means of a cut) is a big improvement.
- Then most Prolog systems will automatically deduce that also `abs_list` is deterministic.

  For the only possible binding patterns `bf` and `bb`.
- Usually, the outermost functor of the first argument is considered: Since it is “`[]`” for the first rule, and “`.`” for the second, always only one of the two rules is applicable (if the first argument is bound).

Cut: Improving Space (7)

- It is also possible to remove unnecessary stack frames at a later point.
- E.g. suppose that `abs` (and thus `abs_list`) remain nondeterministic, and consider the goal:

```
abs_list([-3,7,-4], X), !, p(X).
```

- The call to `abs_list` will leave many stack frames behind, but these are deleted by the cut.

  It is probably better style to avoid the nondeterminism at the place where it occurs. However, one should not use too many cuts, and it might be easier to clean up the stack only at a few places.

Cut: If-Then-Else (1)

- The cut is also used to encode an “if then else”.
- Consider the following predicate:

```
p(X, Y) :- q1(X), !, r1(X, Y).
p(X, Y) :- q2(X), !, r2(X, Y).
p(X, Y) :- r3(X, Y).
```

- This is equivalent to (assuming that q1 and q2 are deterministic):

```
p(X, Y) :- q1(X), r1(X, Y).
p(X, Y) :- \+ q1(X), q2(X), r2(X, Y).
p(X, Y) :- \+ q1(X), \+ q2(X), r3(X, Y).
```

Cut: If-Then-Else (2)

- The formulation with the cut is a bit shorter.

  The difference becomes the bigger, the more cases there are.

- Furthermore, the runtime is shorter: In the version without the cut, `q1(X)` is computed up to three times.

- But removing the cut in first version would completely change the semantics of the program.

  The cut is no longer only an "optimizer hint".

Cut: If-Then-Else (3)

- The logical semantics of programs with negation as failure ("\+") has be extensively studied and there are good proposals.
- I do not know of successful tries to give the cut a clear logical (declarative) semantics.

  The cut can basically be understood only operationally. One problem is that the cut is used for many different purposes, and it might be difficult to automatically discover for which one.
- Pure logic programmers try to avoid the cut, at least when it affects the logic of the program.

Cut: If-Then-Else (4)

- Prolog has an "if-then" operator `->` that can be used to have the advantages of the cut, while making the logical intention clear.
- E.g. one could write the above procedure as

```
p(X, Y) :- q1(X) -> r1(X,Y);
           q2(X) -> r2(X,Y);
           r3(X,Y).
```

- A `->` B has basically the same effect as A, `!`, B.

  However, if there should be further rules about `p`, this cut does not remove the possibility to try these rules. It does remove alternative solutions for A, and it does remove the possibility to try the disjunctive alternatives within the rule.

Cut: Negation

- Conversely, one can implement negation as failure with the cut (`not` is only another name for `\+`):

```
not(A) :- call(A), !, fail.
not(_).
```

- The first rule ensures that if `A` succeeds, `not(A)` fails.
- The second rule makes `not(A)` true in all other cases (i.e. when `A` fails).

  Of course, if `A` should run into an infinite loop, also `not(A)` does not terminate.

Cut: One Solution (1)

- Suppose that email addresses of professors are stored as facts, and that the same person can have several email addresses:

```
prof_email(brass, 'sbrass@sis.pitt.edu').
prof_email(brass, 'brass@acm.org').
prof_email(spring, 'mspring@sis.pitt.edu').
...
```

- The cut can be used to select a single address of a given professor:

```
prof_email(brass, E), !, send_email(E).
```

Cut: One Solution (2)

- Prolog has a built-in predicate `once` that can be used instead of the cut:

```
once(prof_email(brass, E)), send_email(E).
```

- `once` is defined as:

```
once(A) :- call(A), !.
```

- In the example, the following is equivalent:

```
prof_email(brass, E) -> send_email(E).
```

  However, the solution with `once` makes the intention clearer.

Cut: Dangers (1)

- The cut can make programs wrong if predicates are called with unexpected binding patterns.
- E.g. the predicate for the absolute value can also be written as follows (using the cut as in the if-then-else pattern):

```
abs(X,X) :- X >=0, !.
abs(X,Y) :- Y is -X.
```

- Since the second rule is executed only when the first rule is not applicable, it might seem that the test `X =< 0` used earlier is superfluous.

Cut: Dangers (2)

- This is indeed true for the binding pattern `bf`, but consider now the call `abs(3,-3)`!
- In general, the rule is that the cut must be exactly at the point where it is clear that this is the right rule: Not too early and not too late.
- Here the unification must happen after the cut:

```
abs(X,Y) :- X >= 0, !, X = Y.
abs(X,Y) :- Y is -X.
```

- This would work also with binding pattern `bb`.

Cut: Dangers (3)

- Consider this predicate:

```
person(X, male)   :- man(X), !.
person(X, female) :- woman(X).
```

- Since `man` and `woman` are disjoint, the cut was only added to improve the efficiency.
- It works if `person` is called with binding pattern `bf` or `bb`. However, consider what happens if `person` is called with binding pattern `ff`!

  It is interesting that here the more general binding pattern poses a problem, whereas in the `abs` example, the more specific binding pattern is not handled.

Types of Cuts

- Cuts in Prolog programs are usually classified into
  - ◇ Green Cuts: Do not modify the logical meaning of the program, only improve the runtime/space efficiency.

    Some authors also distinguish blue cuts: In this case, a good Prolog system should be able to determine itself that there are no further solutions. Blue cuts are intended only for very simple Prolog systems. “Grue Cuts”: Green or blue cuts.
  - ◇ Red cuts: Modify the declarative meaning of the program.

    Good Prolog programmers try to use red cuts only very seldom.

Cut: Summary, Outlook

- The cut is necessary for efficient Prolog programming, but it destroys the declarative meaning of the programs and can have unexpected consequences.
- The better Prolog implementations get, the less important will be the cut.
- Newer logic programming languages usually try to replace the cut by other constructs that have a more declarative meaning.
- If possible, use `->`, `\+`, `once` instead.
- Use the cut only as last resort.

Overview

1. The Cut and Related Constructs
2. Prolog vs. Pascal
3. Definite Clause Grammars

Prolog vs. Pascal

- "Prolog is different, but not that different."

  This citation is probably from O'Keefe, The Craft of Prolog.

- In general, one can translate a given imperative algorithm (from Pascal, C, etc.) into Prolog.

  The resulting program might be not the best possible program for the task, just as a word-by-word translation from e.g. German to English gives bad English. But at least, if one knows how to solve a problem in an imperative language, one should also be able to write a Prolog program for it.

- The goal of this section is to teach some typical patterns of Prolog programming.

Data Types (1)

Pascal	**Prolog**
integer	integer
real	float
char	ASCII-code (integer) atom
string	list of ASCII-codes atom string (in some Prologs)
file	stream atom (alias, in some Prologs) switching standard IO

Data Types (2)

Pascal	**Prolog**
enumeration type	set of atoms
(variant) record (union/struct in C)	composed term `functor(field1, ..., fieldN)`
array	list set of facts: `a(i, valI)` term: `a(val1, ..., valN)`
pointer	structured terms (e.g., lists) otherwise like array index
—	partial data structures (terms with variables)

Variables (1)

- One can assign a value to a Prolog variable only once.

  This is the biggest difference to imperative programming.

- Afterwards it is automatically replaced everywhere by its value.

  I.e. it ceases to exist as a variable.

- Thus, there is no possibility to assign a different value during normal, forward execution.

  Of course, with backtracking, one can go back to the point where the variable was still unbound. But then all other variable bindings that happend since that point in time are also undone.

Variables (2)

- Thus, one uses a different variable for every value:
  - ◇
    ```
    procedure p(n: integer, var m: integer);
    begin
            m := n * 2;
            m := m + 5;
            m := m * m
    end
    ```
  - ◇
    ```
    p(N, M) :-
            M1  is  N * 2,
            M2  is  M1 + 5,
            M   is  M2 * M2.
    ```

    This is an artificial example. Normally, one would compute the value of m in a single expression.

Variables (3)

- Using a new variable for every assignment is obviously possible for sequential/linear code.
- Loops are formulated in Prolog by recursion, thus one can also get a fresh set of variables for every iteration (see Slide 5-53 for more efficient solution):
  - ◇ **for** i := 1 **to** n **do** writeln(i);
  - ◇
```
loop(N)           :-  loop_body(1, N).
loop_body(I, N)   :-  I > N.
loop_body(I, N)   :-  I =< N, write(I), nl,
                      Next_I is I + 1,
                      loop_body(Next_I, N).
```

Variables (4)

- For variables passed between procedures (in/out-parameters), a Pascal variable is split into two Prolog variables: One for the input value, and one for the output value ("accumulator pair").
  - ◇
    ```
    procedure double(var n: integer);
    begin
            n := n * 2
    end
    ```
  - ◇
    ```
    double(N_In, N_Out) :-
            N_Out  is  N_In * 2.
    ```

Variables (5)

- Global variables in Pascal should be made predicate parameters in Prolog.

  Values that are passed unchanged from predicate to predicate are called "context arguments".

- If there are too many global variables, one can pack them into a structure (composed term) which can be passed as a unit.

  One should declare a predicate for each variable to get/set the value in the structure.

Variables (6)

- One can represent a global variable also with a fact in the dynamic database:
  - ◇ x := x+1
  - ◇ x(X), retract(x(X)), !, X1 is X+1, assert(x(X1)).

    There is always only one fact about the predicate `x` which contains the current value of `X`.

- Some Prolog systems have support for destructive assignments and global variables, but that is very system-dependent.

  In GNU-Prolog, there is, e.g., `g_assign/2`, `g_read/2`, `g_array_size/2`. In SWI-Prolog, see `flag/3`, `setarg/3`. In Sepia/ECLiPSe, see `setval/2`.

Conditions: If, Case (1)

- Example:

  procedure min(i, j: **integer**; **var** m: **integer**);

  begin if i < j **then** m := i **else** m := j **end**

- There are basically three possibilities to translate conditional statements:
  - One rule per case, body contains the complete condition (i.e. the negation of all previous cases):

    ```
    min(I, J, M)  :-  I < J,       M = I.
    min(I, J, M)  :-  \+ (I < J),  M = J.
    ```

    Of course, one would write I >= J instead of \+ (I < J). But not all conditions can be inverted so simply.

Conditions: If, Case (2)

- Translations of **if**-statements, continued:
  - ◇ A rule per case, each starts with the "else if"-condition and a cut:

```
min(I, J, M)  :-  I < J, !,  M = I.
min(I, J, M)  :-             M = J.
```

  - ◇ Using the conditional operators, i.e. a rule with a body of the form (Cond-> Then ; Else):

```
min(I, J, M)  :-  (I < J -> M = I; M = J).
```

  - ◇ If possible, it is best to express the condition in the head of the rule (see next slide).

Conditions: If, Case (3)

- Example (condition in the rule head):
  - ◇
    ```
    procedure p(c: color; var i: integer);
    begin
          case c of red:    i := 1;
                    green:  i := 2;
                    blue:   i := 3;
          end
    end
    ```
  - ◇
    ```
    p(red, I)    :-  I = 1.
    p(green, I)  :-  I = 2.
    p(blue, I)   :-  I = 3.
    ```

    Of course, one would simplify this further to, e.g., p(red, 1).

Conditions: If, Case (4)

- The first solution (complete condition in every rule) is logically cleanest, it also permits to understand each rule in isolation.
- However:
  - ◇ There is a certain duplication of code.
  - ◇ In many Prolog systems, it will leave a choice point behind if the first alternative is chosen.

    Some Prolog systems are intelligent enough to understand that `I < J` and `I >= J` exclude each other, thus no choice point is needed. The solution with the cut or `->` does not have this problem.

Conditions: If, Case (5)

- In general, it is better to use the explicit conditional operator `->` instead of the cut `!`, because this makes the purpose of the cut clear.
  - ◇ However, this makes the syntactical structure of the rules more complicated.
- Most Prolog systems have an index (hash table) over the outermost functor of the first argument.
  - ◇ Thus, if it is possible to code the condition in the rule head (first argument), this will be especially efficient, and no choicepoint will be generated.

Loops: Tail-Recursion (1)

- Loops are usually written as end-recursions.
- One should try to make sure that only a constant amount of memory is needed, not an amount linear in the number of executions of the loop body.
- A Prolog system reuses the memory of a rule invocation when the last body literal is called and there are no more alternatives.

  If necessary, one can use a cut to make clear that other rules are not applicable. It is best when the recursive call is the last literal of the last rule about the predicate.

Loops: Tail-Recursion (2)

- Example (list length):

```
procedure length(l: list; var n: integer);
begin
    n := 0;
    while l <> nil do begin
        n := n + 1;
        l := l ↑.next
    end
end
```

Loops: Tail-Recursion (3)

- Direct Translation to Prolog:

```
length(L, N) :-
        length(L, 0, N).
length(L, N_In, N_Out) :-
        L = [ ], !,
        N_Out = N_In.
length(L, N_In, N_Out) :-
        N_Next is N_In + 1,
        [_|L_Next] = L,
        length(L_Next, N_Next, N_Out).
```

Loops: Tail-Recursion (4)

- Using the rule head for the conditions (makes the cut unnecessary):

```
length(L, N) :-
        length(L, 0, N).
length([], N_In, N_Out) :-
        N_Out = N_In.
length([_|L_Next], N_In, N_Out) :-
        N_Next is N_In + 1,
        length(L_Next, N_Next, N_Out).
```

Loops: Tail-Recursion (5)

- Alternative (elegant, but not tail recursive):

```
length([], 0).
length([_|L_Rest], N) :-
        length(L_Rest, N_Rest),
        N is N_Rest + 1.
```

- In this case, the system must return to a rule invocation after the recursive call.
- Thus, many systems will need memory that is linear in the length of the list. But efficiency is not all!

  The memory will become free after the call to the length predicate. If the lists are not extremely long, the more elegant solution should be preferred.

Loops: Tail-Recursion (6)

- Consider again the for-loop:
  - ◇ **for** i := 1 **to** n **do** writeln(i);
  - ◇
```
loop(N)           :-  loop_body(1, N).
loop_body(I, N)   :-  I > N.
loop_body(I, N)   :-  I =< N, write(I), nl,
                      Next_I is I + 1,
                      loop_body(Next_I, N).
```
- The last call (for I = N+1) probably leaves a choice point behind.
- But then the stack frames of all recursive calls are protected, and the memory complexity is linear.

Loops: Tail-Recursion (7)

- In this case, it might be better to use a cut to make clear that the second rule is no alternative when the first rule is applicable:

```
loop_body(I, N)  :-  I > N, !.
loop_body(I, N)  :-  write(I), nl,
                     Next_I is I + 1,
                     loop_body(Next_I, N).
```

- In this case, one has only the choice between two solutions that are both not nice.

  A good Prolog system might discover that the two conditions I > N and I =< N are mutually exclusive. Then there is no problem.

Loops: Backtracking (1)

- Some loops can also be written with "repeat" and backtracking.
- This is only possible if only the side effect of the loop body is important (input/output, changes to the dynamic database), but no variable bindings must be kept from one iteration to the next.
- If it is possible, it is very efficient.

  Not only the stack frames of predicate invocations are reused, but also all term structures that were built up. This is important for simple Prolog systems that have no garbage collection: There, space on the heap is recycled only upon backtracking. For loops that run an indefinite amout of time (command loops etc.), this should be used.

Loops: Backtracking (2)

-
  ```
  procedure skip(c: character);
  var c1: character;
  begin
          repeat
                  read(c1)
          until c1 = c
  end
  ```

-
  ```
  skip(C)  :-  repeat,
               get_code(C1),
               C1 = C,
               !.
  ```

  The cut is important to avoid the re-execution of the loop if later in the program something fails and backtracking starts.

Loops: Backtracking (3)

- Another typical pattern for a backtracking loop is to iterate over all solutions to a predicate:

```
print_all_composers :-
        composer(FirstName, LastName),
        write(LastName),
        write(', '),
        write(FirstName),
        nl,
        fail.
print_all_composers.
```

The fact at the end ensures that the predicate ultimatively succeees.

Overview

1. The Cut and Related Constructs

2. Prolog vs. Pascal

3. Definite Clause Grammars

Definite Clause Grammars (1)

- In Prolog, one can directly write down grammar rules (of context-free grammars).

  Actually, the grammar formalism is even more expressive, since one can include arbitrary Prolog code as an additional condition for the applicability of a grammar rule.

- A simple preprocessor translates the grammar rules into standard Prolog rules.
- Thus, Prolog has a tool like yacc/bison built-in.

  yacc/bison are standard tools used in compiler construction: Given a context-free grammar (with certain restrictions), they produce a C program that checks the syntactic correctness of the input. One can extend the grammar with program code for processing the input.

Definite Clause Grammars (2)

- However, the goal of the Prolog grammar formalism is not compiler construction, but natural language processing (e.g., machine translation):
  - ◇ There one needs more complicated grammars.
  - ◇ E.g., non-deterministic grammars are possible in Prolog, but not in compiler-construction tools.

    In natural language (e.g., English), there are ambiguous words, phrases, and sentences. These can easily be processed with backtracking in Prolog. In programming languages (e.g., C, Prolog), the meaning of every construct must be completely clear.
  - ◇ However, the efficiency requirements are not so strong, since the inputs are usually not very long.

Definite Clause Grammars (3)

- Comparison with yacc/bison, continued:
  - ◇ In Prolog, arbitrary context-free grammars are possible, in yacc/bison only LALR(1) grammars.

    The condition in compiler construction tools ensures that efficient parsing is possible: The decisions for building the parse tree are done backtrack-free with only a single token lookahead.
  - ◇ In Prolog, grammar rules can be mixed with arbitrary program code. This can contain additional checks for selecting a grammar rule.

    In yacc/bison, one can include C code that is executed when a grammar rule is applied. In this way, one can, e.g. generate output. However, the C code cannot influence the parsing decisions, e.g. choose one of several possible grammar rules.

Definite Clause Grammars (4)

- The grammar on the next slide describes the commands of a simple text adventure game:
    - ◇ Such games are similar to books, in which the reader can give the main actor commands and influence in this way the storyline.

      Therefore, they are called "interactive fiction". Titles like "Zork" by Infocom were very popular.
- In principle, the user can input any English sentence. In practice, most commands
    - ◇ move the player around in the adventure world
    - ◇ apply objects found in certain locations.

Definite Clause Grammars (5)

```
command --> verb, noun_phrase.
command --> [go], direction.
command --> direction.
command --> [quit].
direction --> [north].
direction --> [south].
direction --> [east].
direction --> [west].
verb --> [take].
verb --> [examine].
noun_phrase --> noun.
noun_phrase --> [the], noun.
noun --> [key].
noun --> [lamp].
```

Definite Clause Grammars (6)

- Nonterminal grammar symbols (syntactic categories) are written like standard Prolog predicates.
- Terminal symbols (expected input tokens) are written in `[...]`.
- The syntactic analysis is done with the predicate "**`phrase`**", e.g.
  - ◇ `phrase(command, [take, the, lamp]).` $\longrightarrow$ yes.
  - ◇ `phrase(command, [lamp, the, north]).` $\longrightarrow$ no.

  Of course, one needs more than "yes/no". This is done by attributes of the grammar symbols (predicate arguments). See below.

Implementation (1)

- In principle, every nonterminal symbol N is translated into a Prolog predicate that is true for all lists of input tokens that are derivable from N.
- A naive solution (not the Prolog solution) would generate rules like

```
command(X)          :- append(Y, Z, X),
                       verb(Y), noun_phrase(Z).
command([go|X])     :- direction(X).
command(X)          :- direction(X).
command([quit]).
```

Implementation (2)

- The above solution is too inefficient: Especially the arbitrary splitting of the input list with `append` causes a lot of unnecessary backtracking.
- The real implementation of grammar rules uses a data structure called "difference lists":
  - ◇ E.g. the list `[a, b, c]` is represented by a pair of lists `[a, b, c | X]` and `X`.
  - ◇ A special case is the pair `[a, b, c, d, e]`, `[d, e]`: This also represents the list `[a, b, c]`.

Implementation (3)

- Thus, every nonterminal symbol is translated into a predicate with two arguments:
  - ◇ Input list (total rest of input tokens before the nonterminal symbol is processed).
  - ◇ Output list (rest of input after the nonterminal).
- The difference between both lists are the input symbols derivable from the nonterminal symbol:

```
command(X, Z) :- verb(X, Y), noun_phrase(Y, Z).
command(X, Z) :- X = [go|Y], direction(Y, Z).
command(X, Z) :- direction(X, Z).
command(X, Z) :- X = [quit|Z].
```

Implementation (4)

- I.e. every predicate cuts off from the input list the prefix it can process, and hands the rest to the next predicate.
- The syntax analysis is then done by calling the predicate for the start symbol of the grammar with the complete input list and the empty list as the rest:

```
command([take, the lamp], []).
```

- Thus, the predicate `phrase` is defined as:

```
phrase(Start, Input) :- Goal =.. [Start,Input,[]],
                        call(Goal).
```

Attributes (1)

- Usually, it is not sufficient to know that the input is syntactically correct, but one needs to collect data from the input.
- Therefore, the nonterminal symbols can have arguments (which correspond to attributes in attribute grammars).
- The preprocessor for grammar rules simply extends the given literals by two further arguments for the input and output token lists.

  The given arguments are left untouched.

Attributes (2)

```
command(V,O) --> verb(V), noun_phrase(O).
command(go,D) --> [go], direction(D).
command(go,D) --> direction(D).
command(quit,nil) --> [quit].
direction(n) --> [north].
direction(s) --> [south].
direction(e) --> [east].
direction(w) --> [west].
verb(take) --> [take].
verb(examine) --> [examine].
noun_phrase(O) --> noun(O).
noun_phrase(O) --> [the], noun(O).
noun(key) --> [key].
noun(lamp) --> [lamp].
```

Further Possibilities (1)

- One can include arbitrary Prolog code in the syntax rules.
- It must be written in `{...}` in order to protect it from the rewriting done by the preprocessor.
- E.g. it might be easier to store a list of game objects as facts, and to use only a single grammar rule for nouns:

```
noun(O) --> [O], {object(O, _, _)}.
```

The additional arguments of `object` could be the initial location and the description of the object.

Further Possibilities (2)

- The cut `!`, the disjunction (which can also be written `|`), and the if-then symbol `->` do not need to be included in `{...}`.

  One can also use parentheses `(...)` to structure the alternatives.

- For instance, the optional article before the noun can also be encoded in a single rule:

```
noun_phrase(O) --> ([the] | []), noun(O).
```

- The cut can help to improve the efficiency of the syntax analysis.

Further Possibilities (3)

- The left hand side of the syntax rule can contain a "look-ahead terminal", e.g.

  ```
  p, [a] --> q, [a].
  ```

  means that the production `p --> q` can only be applied if `a` is the next token.

  This is translated to `p(X1,X4) :- q(X1,X2), X2=[a|X3], X4=[a|X3]`, i.e. the look-ahead terminal" is inserted back into the input stream after the rule is processed. In the example, `X2 = X4`, thus the `a` is not consumed.

Efficiency Improvments

- Avoid left recursion.

  This is usually not only inefficient, but wrong: At least for incorrect inputs it easily gets into an endless recursion.

- Think about possible cuts, especially before tail recursions.

- It might be possible to use the Prolog index over the first argument.

```
lookahead(Token), [Token] --> [Token].
stmt --> lookahead(Token), stmt(Token).
stmt(if) --> [if], cond, [then], stmt.
...
stmt(id) --> [id], [':='], expression.
```

Formal Syntax

```
g_rule --> g_head, ['-->'], g_alt.
g_head --> non_terminal, ([','], terminal | []).
g_alt  --> g_if, (['|'], g_alt | []).
g_if   --> g_rhs, (['->'], g_rhs | []).
g_rhs  --> g_item, ([,], g_rhs | []).
g_item --> terminal.
g_item --> non_terminal.
g_item --> variable.
g_item --> ['!'].
g_item --> ['('], g_alt, [')'].
g_item --> ['{'], prolog_goal, ['}'].
non_terminal --> any_callable_prolog_term.
terminal --> ['['], (toks | []), [']'].
toks --> any_prolog_term, ([','], toks | []).
```

Lexical Analysis (1)

- The input to the syntax analysis (parser) is usually a list of word symbols, called tokens.
- Of course, one could also use a list of characters (atoms or ASCII codes).
- However, since the combination of characters to words is simple, a more efficient algorithm (without backtracking) can be used.
- This reduces a long list of characters to a short list of words. Then the more complex algorithm can work on a shorter input.

Lexical Analysis (2)

- The module that is responsible for transforming a sequence of characters into a sequence of word symbols (lexical analysis) is called the scanner.
- The separation of lexical analysis and syntax analysis is also useful because the scanner can suppress
  - ◇ white space between tokens

    Usually, any sequence of spaces, tabulator characters, and line ends is permitted.
  - ◇ comments.
- This simplifies the syntax analysis.

Lexical Analysis (3)

- The following example program reads input characters until the line end.
- It skips spaces and composes sequences of letters to words. Other characters (punctation marks etc.) are treated as one-character tokens.
- The main work is done by a predicate `scan` that gets the current input character as first argument:

```
scanner(TokList) :-
        get_char(C),
        scan(C, TokList).
```

Lexical Analysis (4)

```
scan('\n', []) :- !.
scan(' ', TokList) :- !,
    get_char(C),
    scan(C, TokList).
scan(C, [Word|TokList]) :-
    letter(C), !,
    read_word(C, Letters, NextC),
    name(Word, Letters),
    scan(NextC, TokList).
scan(C, [Sym|TokList]) :-
    name(Sym, [C]),
    get_char(NextC),
    scan(NextC, TokList).
```

Lexical Analysis (5)

- The predicate `read_word` reads a list of letters starting with character `C`. It returns the character `NextC` that follows after the word:

```
read_word(C, [C|MoreC], NextC) :-
    letter(C), !,
    get_char(C2),
    read_word(C2, MoreC, NextC).
read_word(C, [], C).
```

- The predicate `letter` defines which characters can appear in words:

```
letter('a').
...
```