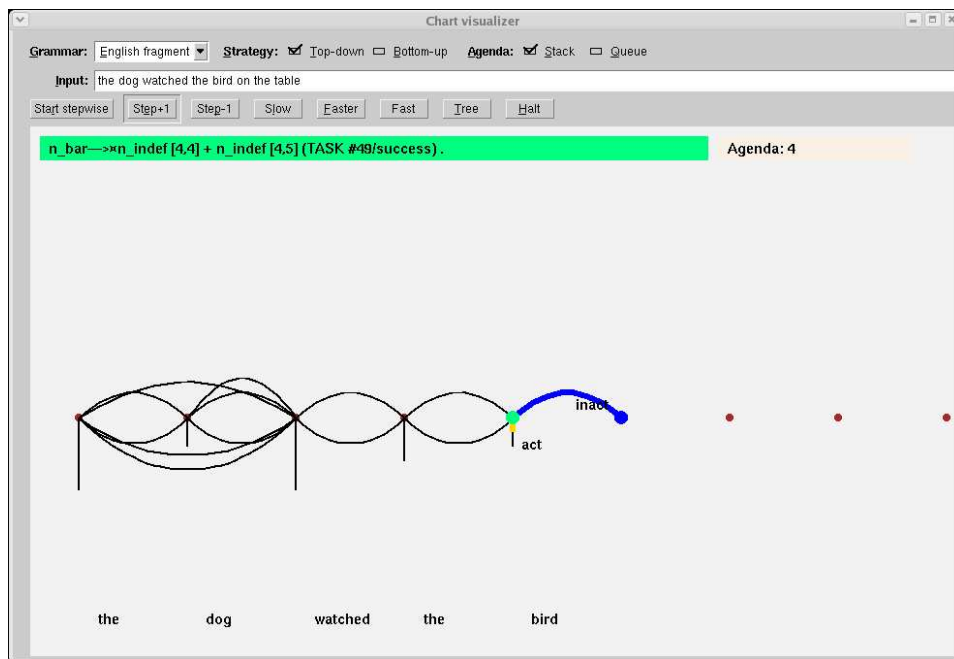


A Chart Parser in Prolog

1 Introduction

This document describes a chart parser implementation in Prolog. It is designed for pedagogical purposes and is equipped with a graphical interface which allows the user to follow the parsing process in the traditional way of drawing charts. The parser is for context-free grammars. The parsing algorithm can be modified in two ways: First, a predictive and a bottom-up strategy are available. Secondly, the chart can be treated as either a queue or a stack.

The implementation is written in SWI-Prolog and the graphical interface uses its XPCE package (see <http://www.swi-prolog.org/packages/xpce/>).



2 The Data Structures

2.1 Grammars

The parsing system works with context-free grammars. A distinction is made between lexical categories (preterminal symbols) and phrasal ones, as in most natu-

ral language applications. The categories are Prolog atoms in the present examples. A grammar is a lists of terms, each of which are of one of the following kinds:

Start symbol clause: A term of the form `start_symbol(X)` states that X is the start symbol. There should be one such term in a grammar.

Preterminals (lexical categories): A term of the form `preterminals(X)` states that X is the list of preterminal symbols, i.e. lexical categories. There should be one such term in a grammar.

Phrase structure rules: A term of the form `L ---> R` is a phrase structure rule stating that L can be rewritten as R. L is a single non-preterminal and non-terminal symbol and R is a list of preterminal and other non-terminal symbols. Operator declaration: `:-op(900,xfx,--->)`

Lexical entries: A term of the form `W : : C` states that the terminal symbol (word) W is of lexical category (preterminal symbol) C. Operator declaration: `:-op(900,xfx,::)`

Example grammars are stored as `sample_grammar/2` facts. A fact `sample_grammar(Name, Grammar)` gives a certain grammar, Grammar, the name Name. When a grammar is put to use, the `start_symbol/1`, `preterminals/1`, `--->/2`, and `::/2` terms are entered as dynamic predicate facts to the Prolog database.

This is an example of a grammar, embedded in a `sample_grammar/2` fact.

```
% sample_grammar(+Name,-Grammar)
% Grammar is a grammar by the name of Name.

sample_grammar(english_fragment,
               [start_symbol(s),

                preterminals([conj,pn,det,n_indef,
                              iv,tv,prep]),

                s ---> [np,vp],
                s ---> [s,conj,s],
                np ---> [pn],
                np ---> [det,n_bar],
                np ---> [np,conj,np],
                n_bar --->[n_indef],
                n_bar --->[n_bar,pp],
                vp ---> [iv],
                vp ---> [tv,np],
                vp ---> [vp,conj,vp],
```

```

pp ---> [prep,np],

det::a,
det::the,
n_indef::bird,
n_indef::cat,
n_indef::dog,
n_indef::box,
n_indef::table,
pn::john,
pn::mary,
iv::slept,
tv::watched,
conj::and,
prep::in,
prep::on]).

```

A sample grammar is selected and stored in the Prolog database by means of the procedure `select_grammar/1`:

```

% select_grammar(+Name)
%   Retracts the dynamic predicates start_symbol/1,
%   preterminals/1, --->/2, and ::/2, if they are
%   previously defined, and asserts the facts
%   stated in terms of these four predicates
%   found in the grammar by the name Name,
%   as defined by the sample_grammar/2 predicate.

select_grammar(Name):-
    retractall(start_symbol(_)),
    retractall(preterminals(_)),
    retractall((--->_)),
    retractall(_::_),
    sample_grammar(Name,Clauses),
    assert_clauses(Clauses).

assert_clauses([]).

assert_clauses([Clause|Clauses]):-
    assertz(Clause),
    assert_clauses(Clauses).

```

2.2 Syntactic descriptions

Syntactic descriptions are syntax trees here. They are represented as follows: A leaf, a word being of a certain category, is represented as `Category-Word`. Trees with subtrees are represented as `Mother/Daughter_list`, where `Mother` is the category of the mother node, and `Daughter_list` is the list of daughter subtrees.

We need the following three construction predicates for syntax trees: First, there is one for minimal complete trees, i.e. for trees with a preterminal dominating a terminal node. Secondly, we must be able to build trees without daughters. Thirdly, we need a way of adding a new daughter subtree to a tree.

```
% make_leaf(+Category,+Word,-Tree)
%   Makes a minimal complete tree, Tree, with a
%   preterminal, Category, dominating a terminal node, Word.

make_leaf(Category,Word,Category-Word).

% make_tree_without_daughters(+Category,-Tree)
%   Makes a tree, Tree, without daughters, with Category
%   labelling the mother node.

make_tree_without_daughters(Category,Category/[]).

% add_daughter_to_tree(+Tree_in,+Daughter,-Tree_out)
%   Makes a new tree, Tree_out, formed by adding the
%   tree, Daughter, as a new rightmost daughter to the
%   tree Tree_in.

add_daughter_to_tree(Mother/Daughter_list,
                    New_daughter,
                    Mother/New_daughter_list):-
    append(Daughter_list,[New_daughter],New_daughter_list).
```

2.3 Edges

As usual in a chart parser, complete and incomplete analyses are stored as so-called **edges**. The locations of words and substrings are defined by pairs of positions (the leftmost and the rightmost one of the expression). Positions are non-negative integers: 0 is the beginning of the input string. 1 separates the first word from the second, etc. Those edges corresponding to partially recognized expressions are said to be *active*, while the *inactive* edges represent completely recognised substrings. An active edge is defined by a start position, an end position, a syntactic description, and a list of not yet found constituents. This list defines what is required to complete the assemblage of an expression. These lists derive from the right-hand sides of the grammar rules. On in-

active edges, this list of not yet found constituents is empty. So, an edge is a term `edge(Position1,Position2,Category,To_find_list,Tree)`, where `Position1` and `Position2` are positions, $Position1 \leq Position2$, `Category` is a category, `To_find_list` is a list of categories, and `Tree` is a syntactic description. `Tree` describes the constituent as far as it has been recognized, i.e. it only contains those constituents that actually are within the span of the edge. When `To_find_list=[]`, the edge is inactive. On an active edge, `To_find_list=[_ _]`.

2.4 Charts

A chart is represented as a Prolog list of edges. We need to be able to create an empty chart and to create a new chart by adding an edge to an existing chart. We also need to retrieve edges from charts, and to test whether a certain edge is a member of a given chart.

```
% empty_chart(?Chart)
%   Chart is the empty chart.

empty_chart([]).

% edges_not_already_in_chart(+Edges1,+Chart,-Edges2)
%   Finds those edges, Edges2, among the edges in the
%   list Edges1 which are not already found in Chart.

edges_not_already_in_chart([],_,[]).

edges_not_already_in_chart([Edge|Edges1],Chart,Edges2):-
    edge_in_chart(Edge,Chart),
    edges_not_already_in_chart(Edges1,Chart,Edges2).

edges_not_already_in_chart([Edge|Edges1],Chart,[Edge|Edges2]):-
    \+edge_in_chart(Edge,Chart),
    edges_not_already_in_chart(Edges1,Chart,Edges2).

% add_edge_to_chart(+Edge,+Chart_in,-Chart_out)
%   Chart_out is the chart formed by adding Edge
%   to Chart_in.

add_edge_to_chart(Edge,Chart_in,Chart_out):-
    append(Chart_in,[Edge],Chart_out).

% edge_in_chart(?Edge,+Chart)
%   Edge is a member of Chart.
```

```
edge_in_chart (Edge, Chart) :-  
    member (Edge, Chart) .
```

2.5 Tasks

Tasks correspond to the basic steps of the parsing process. They are stored on the so-called agenda, where they wait for execution. In this implementation the concept of task has been used for a wider range of purposes than in most presentations of chart parsing. Both rule activation and lexical look-up are requested by special kinds of task.

- Tasks motivated by the “fundamental rule of chart parsing”, i.e. those resulting from the meeting of an active edge and an inactive one are represented by terms of the form `fundamental_task(Edge1, Edge2)`. `Edge1` is the active edge and `Edge2` is the inactive one. (This is the traditional concept of task.)
- Words in the input string are looked up when tasks of the form `look_up(Position, Word)` are executed. `Word` is the word form found in the string and `Position` is its left position.
- Rule predictions are triggered by tasks of the form `predict(Position, Category)`. When such a task is executed, all rules for building expressions of category `Category` are predicted at position `Position`. This means that for each rule `Category ---> Right`, an edge `edge(Position, Position, Category, Right, Category/[])` is added to the chart.
- Bottom-up rule activations are triggered by tasks of the form `bu_activate(Pos1, Pos2, Category)`. It requires that all rules such that the first symbol on the right hand side is `Category` be activated at position `Pos1`. (`Pos1-Pos2` is the span of the actual `Category` expression that has motivated the task, but the `Pos2` value is actually of no consequence.) When such a task is executed, for each rule `Left ---> [Category|Categories]`, an edge `edge(Pos1, Pos1, Left, [Category|Categories], Left/[])` is added to the chart.

2.6 The Agenda

The agenda is part of the basic control mechanism of a chart parser. It holds those tasks which are to be executed. The agenda is simply a list of tasks. Both a “last-in, first-out” (queue) and a “first-in, first-out” (stack) policy can be selected. This choice determines the order in which the tasks are executed, but does not influence the final outcome of the parsing process (the chart when the agenda is empty).

```

% empty_agenda(?Agenda)
%   Agenda is the empty agenda.

empty_agenda([]).

% retrieve_task_from_agenda(-Task,+Agenda_in,-Agenda_out)
%   Task is the first Task on Agenda_in. Agenda_out
%   is the agenda that remains after Task have been removed
%   from Agenda_in.

retrieve_task_from_agenda(Task,[Task|Agenda_out],Agenda_out).

% agenda_size(+Agenda,-Size)
%   Size is the number of tasks on Agenda.

agenda_size(Agenda,Size):-
    length(Agenda,Size).

```

The dynamic predicate `store_tasks_on_agenda/3` is used to add new tasks to the agenda. There are two versions of the predicate, which implement, respectively, a queue and a stack policy. The predicate `select_agenda_policy/1` selects dynamically which version will be used.

```

% select_agenda_policy(+Policy)
%   Selects the agenda policy Policy, Policy=queue
%   or Policy=stack, by redefining the dynamic
%   predicate store_tasks_on_agenda/3. Both
%   policies are implemented by means of append/3,
%   the difference being in which end the new tasks
%   are appended.

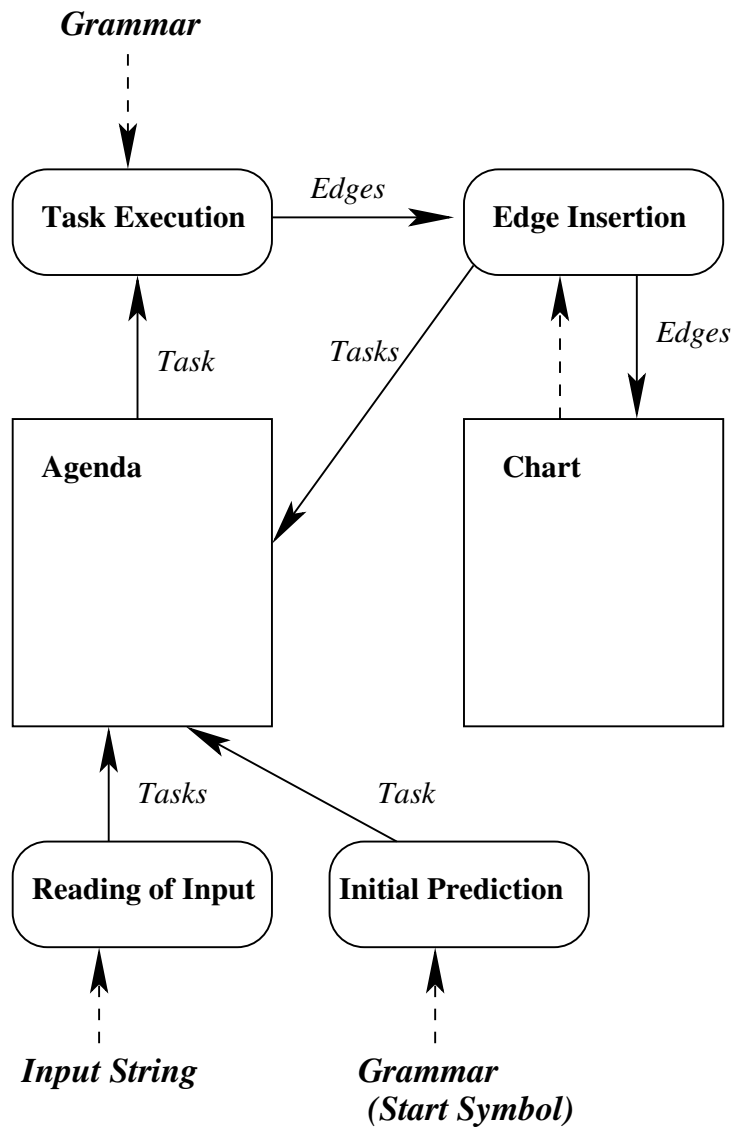
select_agenda_policy(queue):-
    retractall(store_tasks_on_agenda(_,_,_)),
    assert((store_tasks_on_agenda(New_tasks,
                                Agenda_in,
                                Agenda_out):-
        append(Agenda_in,New_tasks,Agenda_out))).

select_agenda_policy(stack):-
    retractall(store_tasks_on_agenda(_,_,_)),
    assert((store_tasks_on_agenda(New_tasks,
                                Agenda_in,
                                Agenda_out):-
        append(New_tasks,Agenda_in,Agenda_out))).

```

3 The Parsing Mechanisms

The parsing algorithm is the result of interactions among a number of submechanisms. The most important ones are the execution of tasks and the scheduling of new tasks due to insertions of new edges into the chart. This is an overview in a procedural style:



Initial steps: The main control mechanism involves defining the initial tasks. These involve:

Lexical look-up tasks: The input string is transformed into a set of look-up tasks, which are put on the agenda.

Initial prediction task: If a top-down strategy is selected, an initial prediction task must be put on the agenda.

Agenda handling: New tasks must be scheduled on the agenda and each task on the agenda must be executed.

Task execution: The execution of tasks produces a number, possibly zero, of edges, which, if not already there, are added to the chart.

Edge insertion: The insertion of edges into the chart may lead to new tasks being produced. Rules are activated in a top-down or bottom-up fashion, depending on which strategy has been selected. Fundamental rule tasks are scheduled whenever an active and an inactive edge meet.

3.1 The initial steps

The `record_event/1` calls record the basic steps of the parser for possible use by the visualization tool. The `start_recording_events/0` procedure initialises the recording process. These procedures do not play any role in the actual parsing computations.

The main procedure of the parser is `parse/6`:

```
% parse(+Input_string,+Grammar,+Strategy,+Agenda_policy,
%       -Number_of_words_in_input,-Final_chart)
% Builds the final chart given an input string,
% a grammar (name), a strategy, and an agenda policy.
% It also counts the number of words in the input
% string.

parse(Input_string,
      Grammar,
      Strategy,
      Agenda_policy,
      Number_of_words_in_input,
      Final_chart):-
    %% This is for the 'recording' process:
    start_recording_events,
    %% Select grammar, strategy, and agenda policy.
    select_grammar(Grammar),
    select_strategy(Strategy),
    select_agenda_policy(Agenda_policy),
    %% Start from an empty agenda.
    empty_agenda(Fresh_agenda),
    record_event(nodes(Input_string)),
    %% Store the lexical look-up tasks on the agenda.
```

```

lexical_tasks(Input_string,0,Number_of_words_in_input,
              Lexical_tasks),
store_tasks_on_agenda(Lexical_tasks,Fresh_agenda,
                    Lexical_agenda),
    %% Store the initial prediction task on the agenda.
    %% (initial_prediction_tasks/1 is strategy-dependent.)
initial_prediction_tasks(Prediction_tasks),
store_tasks_on_agenda(Prediction_tasks,Lexical_agenda,
                    Agenda_with_prediction),
    %% Process (and update) the successive agendas
    %% until we arrive at an empty one.
    %% Start from an empty chart.
empty_chart(Fresh_chart),
process_agenda(Agenda_with_prediction,1,
              Fresh_chart,Final_chart).

```

First, a grammar, a parsing strategy ('top-down' or 'bottom-up') and an agenda policy (stack or queue) are selected. These selections correspond to (re)definitions of dynamic predicates. Then, starting from an empty agenda, the lexical look-up tasks produced from the input string are put on the agenda. If the 'top-down' strategy is selected, the next step is to put the initial predication task on the agenda. The successive agendas are then processed until an empty agenda is reached. The initial chart is an empty one.

3.2 Agenda processing

The parsing process is driven by the predicate `process_agenda/5`. It processes the successive agendas, starting from a given one, and successively updates a given chart. The fifth argument of `process_agenda/5` is the final chart, reached at the termination of the parsing process. The second argument gives the number of the first task on the given agenda. It is only used for reporting purposes.

```

% process_agenda(+Agenda,+Task_number,+Chart_in,-Chart_out)
%   Processes the successive agendas, starting from Agenda
%   and the chart Chart_in, until an empty agenda is
%   reached, producing the final chart Chart_out.
%   Task_number gives a number to the first task on the
%   agenda. It is only used for reporting.

```

```

process_agenda(Agenda,_Task_number,Chart,Chart):-
    empty_agenda(Agenda).

```

```

process_agenda(Agenda,Task_number,Chart_in,Chart_out):-
    retrieve_task_from_agenda(Task,Agenda,Agenda_mediating),

```

```

execute_task(Task,Task_number,Chart_in,Chart_mediating,
            Agenda_mediating,Agenda_out),
Task_number_next is Task_number + 1,
process_agenda(Agenda_out,Task_number_next,
            Chart_mediating,Chart_out).

```

3.3 Edge insertion

The procedure `add_edges_and_update_agenda/6` inserts edges (new ones) into the chart, and updates the agenda with tasks that have to be put on the agenda due to these edges. These tasks are either fundamental ones, or rule activation tasks, as specified by the current parsing strategy.

```

% add_edges_and_update_agenda(+Edges,+Agenda_in,-Agenda_out,
%                             +Chart_in,-Chart_out,+Origin)
% Adds the edges in the list Edges to Chart_in giving us
% Chart_out and adds the new tasks motivated by the edges
% to Agenda_in giving us Agenda_out. Origin is a term
% describing the origin of the edge. It is only used for
% reporting purposes, and plays no role in the parsing
% process.

```

```

add_edges_and_update_agenda([],Chart,Chart,Agenda,Agenda,_).

```

```

add_edges_and_update_agenda([Edge|Edges],Chart_in,Chart_out,
                            Agenda1,Agenda4,Origin):-
    add_edge_to_chart(Edge,Chart_in,Chart_mediating),
    new_fundamental_tasks(Edge,Chart_in,Fundamental_tasks),
    % Rule activation, depending on strategy:
    rule_activation_tasks(Edge,Activation_tasks),
    store_tasks_on_agenda(Fundamental_tasks,Agenda1,Agenda2),
    store_tasks_on_agenda(Activation_tasks,Agenda2,Agenda3),
    record_event(edge_addition(Edge,Origin,Agenda1,
                              Agenda2,Activation_tasks)),
    add_edges_and_update_agenda(Edges,Chart_mediating,
                              Chart_out,Agenda3,Agenda4,
                              Origin).

```

3.4 Lexical task scheduling

The input string is initially consumed by being converted into a set of lexical look-up tasks, which are stored on the agenda. The look-up of the individual words is thus a matter of agenda bookkeeping.

```

% lexical_tasks(+String,+Current_position,-End_position,-Tasks)

```

```

% Transforms the input String from Current_position into a
% set of lexical look-up Tasks. Current_position is the
% position directly to the left of the head of String.
% End_position is the number of words read from position 0.

lexical_tasks([],Number_of_words,Number_of_words,[]).

lexical_tasks([Word|Words],Current_position,Number_of_words,
              [look_up(Current_position,Word)|Tasks]):-
    Next_position is Current_position + 1,
    lexical_tasks(Words,Next_position,Number_of_words,Tasks).

```

3.5 Fundamental task scheduling

Each new fundamental task must be put on the agenda. These are found by means of `new_fundamental_tasks/3`.

```

% new_fundamental_tasks(+New_edge,+Chart,-New_tasks)
% Returns the list of fundamental tasks New_tasks
% formed by New_edge and the edges in the Chart.

new_fundamental_tasks(Inactive_edge,Chart,New_tasks):-
    Inactive_edge=edge(Position,_,_,[],_),
    findall(fundamental_task(Active_edge,Inactive_edge),
            (Active_edge=edge(_,Position,_,[_|_],_),
             edge_in_chart(Active_edge,Chart))),
    New_tasks).

new_fundamental_tasks(Active_edge,Chart,New_tasks):-
    Active_edge=edge(_,Position,_,[_|_],_),
    findall(fundamental_task(Active_edge,Inactive_edge),
            (Inactive_edge=edge(Position,_,_,[],_),
             edge_in_chart(Inactive_edge,Chart))),
    New_tasks).

```

3.6 Rule activation tasks and the parsing strategies

The parsing strategy is determined by the definitions of the two dynamic predicates `initial_prediction_tasks/1` and `rule_activation_tasks/2`. These definitions are thus embedded within `assert/1` clauses in the code. The first one `initial_prediction_tasks/1` gives us the initial prediction necessary for a top-down parsing procedure. This is its definition under the top-down strategy. It states that the start symbol is to be predicted from the leftmost position.

```

initial_prediction_tasks([predict(0,Start_symbol)]) :-

```

```
start_symbol(Start_symbol).
```

The definition under the bottom-up strategy makes sure that no prediction is made initially:

```
initial_prediction_tasks([]).
```

The relation `rule_activation_tasks/2` gives us the set of rule activation tasks triggered by the insertion of an edge in the chart. This is its top-down definition. First clause: it does not make any predictions from inactive edges. Second clause: it does not predict lexical categories. Third clause: An active edge from `Position1` to `Position2` with a non-lexical `Category` as the first symbol on its to-find list leads to a prediction of a `Category` constituent from `Position2`.

```
rule_activation_tasks(edge(_,_,_, [], _), []).
```

```
rule_activation_tasks(edge(Position1,Position2,_,
                          [Category|_],_), []):-
    preterminals(Lexical_categories),
    member(Category, Lexical_categories).
```

```
rule_activation_tasks(edge(Position1,Position2,_,
                          [Category|_],_),
                      [predict(Position2,Category)]):-
    preterminals(Lexical_categories),
    \+member(Category, Lexical_categories)).
```

The bottom-up version of `rule_activation_tasks/2` is defined by two clauses: First, active edges does not trigger bottom-up rule activation tasks. Secondly, when an inactive edge, `edge(Position1,Position2,Category, [], -)`, is added to the chart a `bu_activate(Position1,Position2,Category)` task is to be put on the agenda. (As mentioned previously, the `Position2` information is not used in the parsing process.)

```
rule_activation_tasks(edge(_,_,_, [_|_], _), []).
```

```
rule_activation_tasks(edge(Position1,Position2,Category, [], _),
                      [bu_activate(Position1,Position2,
                                   Category)]).
```

The `select_strategy/1` is responsible for the `assert/1`-ing of the definitions of the strategy predicates `initial_prediction_tasks/1` and `rule_activation_tasks/2`:

```

% select_strategy(+Strategy),
%   Selects Strategy, where Strategy='top-down' or
%   Strategy='bottom-up' by redefining the dynamic
%   predicates initial_prediction_tasks/1 and
%   rule_activation_tasks/2.

select_strategy('top-down'):-
    retractall(initial_prediction_tasks(_)),
    retractall(rule_activation_tasks(_,_)),
    %% Predict the start symbol at position 0:
    assert((initial_prediction_tasks([predict(0,Start_symbol)]):-
        start_symbol(Start_symbol))),
    %% Inactive edges do not trigger predictions:
    assert((rule_activation_tasks(edge(_,_,[ ],_),[ ])),
    %% Do not predict lexical categories:
    assert((rule_activation_tasks(edge(Position1,Position2,_,
        [Category|_],_),[ ]):-
        preterminals(Lexical_categories),
        member(Category,Lexical_categories))),
    %% Predict non-lexical categories:
    assert((rule_activation_tasks(edge(Position1,Position2,_,
        [Category|_],_),
        [predict(Position2,Category)]):-
        preterminals(Lexical_categories),
        \+member(Category,Lexical_categories))).

select_strategy('bottom-up'):-
    retractall(initial_prediction_tasks(_)),
    retractall(rule_activation_tasks(_,_)),
    %% Do not make any (initial) prediction:
    assert(initial_prediction_tasks([ ])),
    %% Active edges do not trigger bottom-up rule activation:
    assert(rule_activation_tasks(edge(_,_,[_|_],_),[ ])),
    %% Inactive edges trigger bottom-up rule activation:
    assert(rule_activation_tasks(edge(Position1,
        Position2,
        Category,[ ],_),
        [bu_activate(Position1,
            Position2,
            Category)]))).

```

3.7 Task execution

The task execution procedure `execute_task/6` is responsible for the basic steps of processing in the parser. See section 2.5 for a listing of the various kinds of task. There is one clause for each kind of task in the definition of `execute_task/6`, except for the fundamental tasks, which are treated by two clauses: One for the success case, and one for the failure case.

```
% execute_task(+Task,+Number,+Chart_in,-Chart_out,
%             +Agenda_in,-Agenda_out,)
% Executes the Task with the chart and the agenda
% being Chart_in and Agenda_in, giving us Chart_out
% and Agenda_out.

% This clause is for the execution of successful fundamental tasks.

execute_task(fundamental_task(Edge1,Edge2),
             Task_number,Chart_in,Chart_out,Agenda_in,Agenda_out):-
    Edge1=edge(Position1,Position2,Active_category,
               [First_to_find|Tail_to_find],Active_tree),
    Edge2=edge(Position2,Position3,Inactive_category,
               [],Inactive_tree),
    First_to_find=Inactive_category, %% succeeds
    add_daughter_to_tree(Active_tree,Inactive_tree,New_tree),
    edges_not_already_in_chart([edge(Position1,Position3,
                                     Active_category,Tail_to_find,
                                     New_tree)]),
    Chart_in,New_edges),
    record_event(fundamental_task(Edge1,Edge2,Task_number,
                                  Agenda_in,success,New_edges)),
    add_edges_and_update_agenda(New_edges,Chart_in,Chart_out,
                                Agenda_in,Agenda_out,task).

% This clause is for the execution of failing fundamental tasks.
% (The execute_task/6 call succeeds anyway.)

execute_task(fundamental_task(Edge1,Edge2),
             Task_number,Chart,Agenda,Agenda):-
    Edge1=edge(_,_,[First_to_find|_],_),
    Edge2=edge(_,_ ,Inactive_category,_),
    \+(First_to_find=Inactive_category),
    record_event(fundamental_task(Edge1,Edge2,Task_number,
                                  Agenda,failure,[])).

% This clause is for lexical look-up.
```

```

execute_task(look_up(Position,Word),Task_number,
             Chart_in,Chart_out,Agenda_in,Agenda_out):-
    Next_position is Position + 1,
    findall(edge(Position,Next_position,Category,[],Tree),
            (Category::Word,
             make_leaf(Category,Word,Tree))),
            Edges),
edges_not_already_in_chart(Edges,Chart_in,New_edges),
record_event(look_up(Position,Word,Task_number,
                    Edges,New_edges)),
add_edges_and_update_agenda(New_edges,Chart_in,Chart_out,
                            Agenda_in,Agenda_out,
                            look_up(Position,Word)).

% This clause is for rule prediction. It will only be put to
% use under the top-down strategy.

execute_task(predict(Position,Category),Task_number,
             Chart_in,Chart_out,Agenda_in,Agenda_out):-
    findall(edge(Position,Position,Category,RHitems,Tree),
            (Category-->RHitems,
             make_tree_without_daughters(Category,Tree))),
            Edges),
edges_not_already_in_chart(Edges,Chart_in,New_edges),
record_event(predict(Position,Category,Task_number,
                    Edges,New_edges)),
add_edges_and_update_agenda(New_edges,Chart_in,Chart_out,
                            Agenda_in,Agenda_out,
                            predict(Position,Category)).

% This clause is for bottom-up rule activation. It will only
% be put to use under the bottom-up strategy.

execute_task(bu_activate(Position1,Position2,Category),
             Task_number,Chart_in,Chart_out,
             Agenda_in,Agenda_out):-
    findall(edge(Position1,Position1,LHitem,[Category|RHitems],
                Tree),
            (LHitem-->[Category|RHitems],
             make_tree_without_daughters(LHitem,Tree))),
            Edges),
edges_not_already_in_chart(Edges,Chart_in,New_edges),
record_event(bu_activate(Position1,Position2,Category,

```

```

                                Task_number,Edges,New_edges)),
add_edges_and_update_agenda(New_edges,Chart_in,Chart_out,
                                Agenda_in,Agenda_out,
                                bu_activate(Position1,Position2,
                                            Category)).

```

4 How to use the parser?

The chart parsing procedures, documented in detail here, are found in the file `chartparser.pl`. The graphical interface and a very simple textual one reside in the file `parser_interface.pl`. Sample grammars are put in the file `grammars.pl`. If the three files are placed in the same directory, all three of them will be loaded into the SWI-Prolog system when the file `parser_interface.pl` is consulted.

The files are found at URL:

```

http://stp.ling.uu.se/~matsd/prolog\_chartparser/

```

The parser may be tested without the graphical interface: A `test_parser(+Input_string,+Grammar,+Strategy,+Agenda_policy)` call builds the final chart given an input string, a grammar (name), a strategy, and an agenda policy. It reports the numbers of active and inactive edges, and prints the start symbol analyses spanning the entire input string.

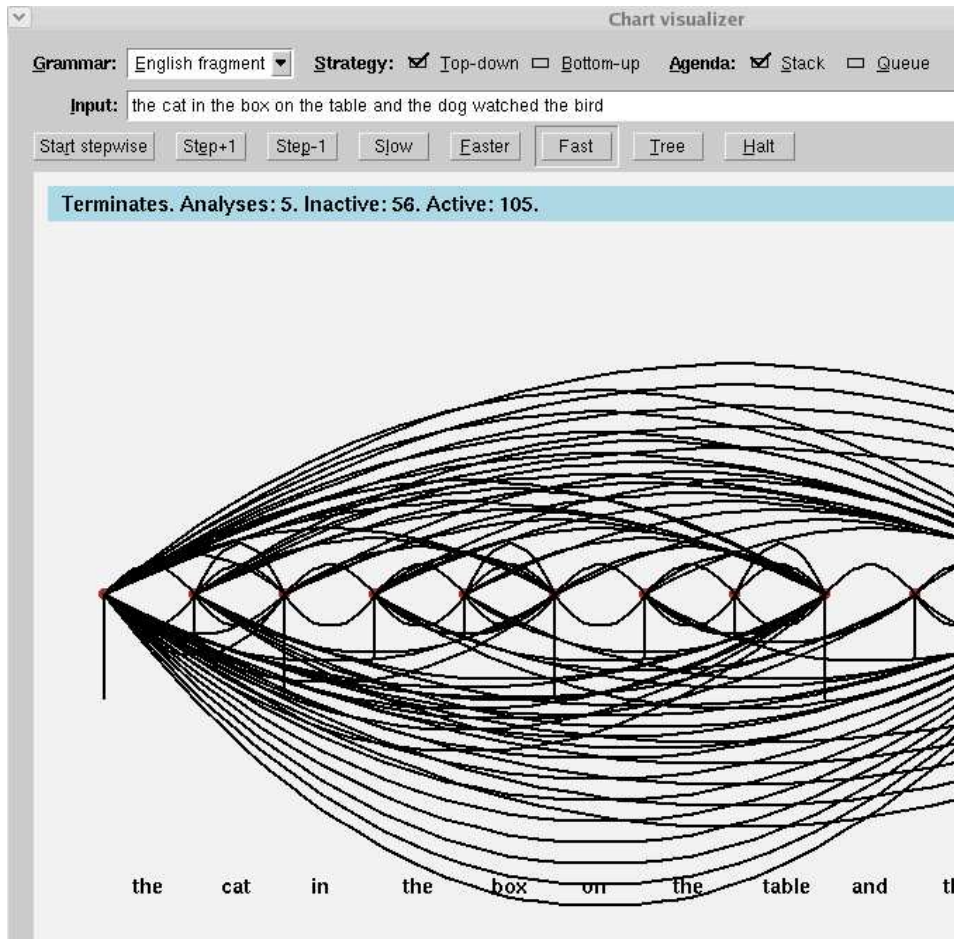
```

?- test_parser([the,dog,slept],english_fragment,'top-down',stack).
Number of active edges: 17
Number of inactive edges: 7
Analyses:
[s/[np/[det-the, n_bar/[n_indef-dog]], vp/[iv-slept]]]
Yes

```

A `dialog/1` call opens the graphical interface window. The window that ap-

pears will look something like this (with a chart drawn):



The grammar (alternatives from the predicate `sample_grammar/2`), the parsing strategy (top-down or bottom-up), and the agenda policy (stack or queue) are selected by means of the drop-down menu and the radio buttons at the top. The input string is entered in a text input field. The Start stepwise option parses the string and prepares the scene for a stepwise examination of the parsing process. The Step+1 button takes the picture one step forward and the Step-1 button takes it one step backward. The buttons Slow, Less slow, Fast shows the parsing process as a kind of animation from start to end.