The Phoenix Parser User Manual

Introduction

The Phoenix parser is designed for development of simple, robust Natural Language interfaces to applications, especially spoken language applications. Because spontaneous speech is often ill-formed and because the recognizer will make recognition errors, it is necessary that the parser be robust to errors in recognition, grammar and fluency. This parser is designed to enable robust parsing of these types of input.

Phoenix parses each input utterance into a sequence of one or more semantic frames. The developer must define a set of frames and provide grammar rules that specify the word strings that can fill each slot in a frame.

Release Configuration

The Phoenix distribution contains the directories:

- **ParserLib** - The parse library functions which are compiled into the archive library libparse.a

- **Grammars** – A set of task independent semantic grammars. These grammars can be compiled in with the task specific grammars for an application. This is normally done with the *compile* script (see *Compiling Grammars* section)

- **Scripts** – Scripts to pack grammars into a single file and to generate a list of net names contained in grammar files. These scripts are called by the *compile* script.

- **Example** – An example frames file and grammars. See the *Tutorial Example* section

- **Server –** Code for Galaxy HUB server interface. This code will not compile without the GALAXY architecture installation, bet serves as a guide to hoe to implement a GALAXY parse server.

- **Doc** – Contains this file.

Theory of Operation

The Phoenix parser maps input word strings onto a sequence of semantic frames. A Phoenix frame is a named set of slots, where the slots represent related pieces of information. Figure 1 shows an example frame for queries for flight information. Each slot has an associated Context-Free Grammar that specifies word string patterns that match the slot. The grammars are compiled into Recursive Transition Networks (RTNs). An example parsed frame is shown in Figure 2. When filled in by the parser, each slot contains a semantic parse tree for the string of words it spans. The root of the parse tree is the slot name.

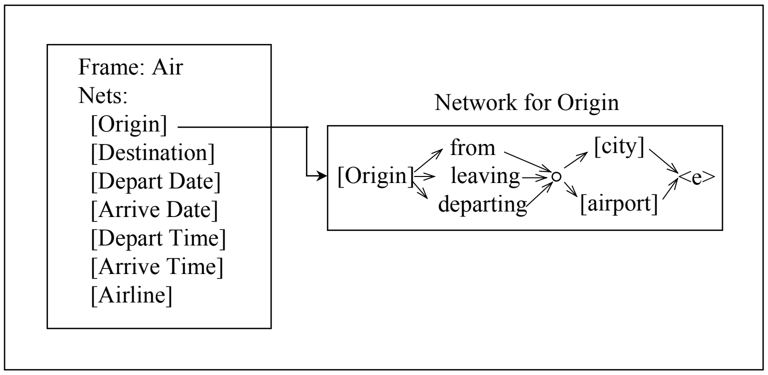
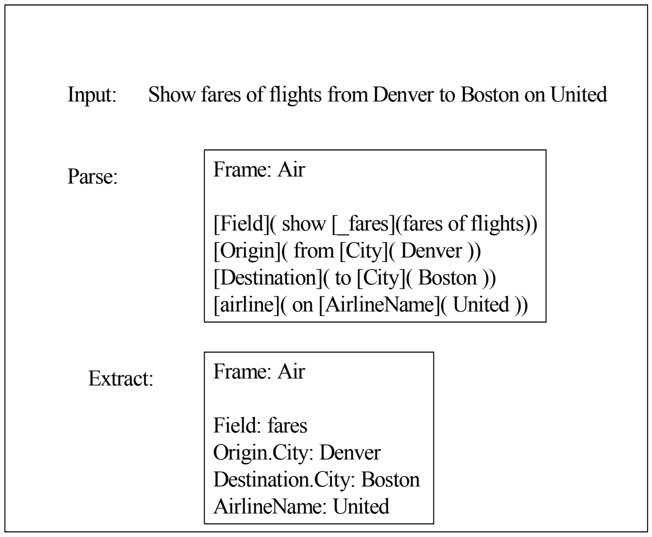


Figure 1
Example Frame



Input: Show fares of flights from Denver to Boston on United

Parse:

Frame: Air

[Field](show [_fares](fares of flights))
[Origin](from [City](Denver))
[Destination](to [City](Boston))
[airline](on [AirlineName](United))

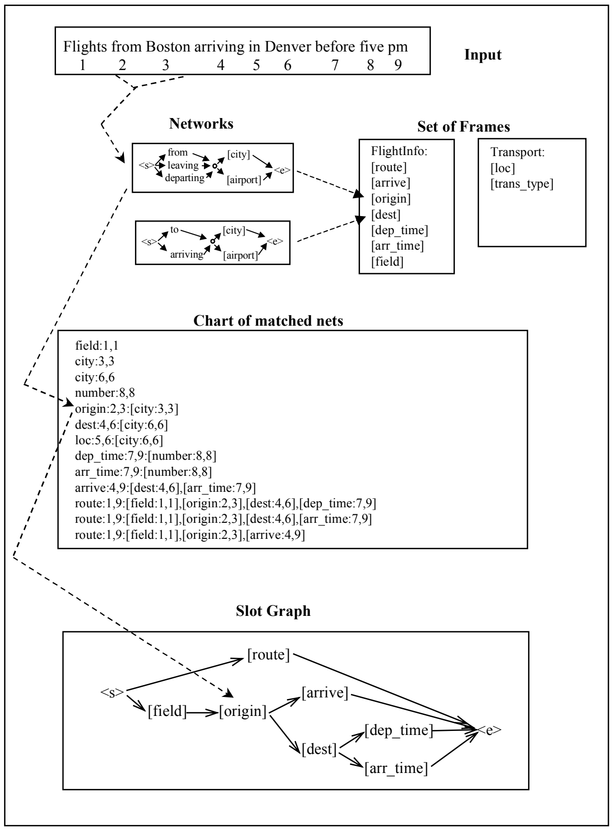
Extract:

Frame: Air

Field: fares
Origin.City: Denver
Destination.City: Boston
AirlineName: United

Figure 2
Example Parsed Output

The parsing process is shown in Figure 3. In a search algorithm very similar to the acoustic match producing a word graph, grammars for slots are matched against a word string to produce a slot graph. The set of active frames defines a set of active slots. Each slot points to the root of an associated Recursive Transition Network. These networks are matched against the input word sequence by a top-down Recursive Transition Network chart parsing algorithm.


Input
Flights from Boston arriving in Denver before five pm
1 2 3 4 5 6 7 8 9
Networks
<s> from leaving departing [city] [airport] <e>
<s> to arriving [city] [airport] <e>
Set of Frames
FlightInfo: [route] [arrive] [origin] [dest] [dep_time] [arr_time] [field]
Transport: [loc] [trans_type]
Chart of matched nets
Slot Graph
<s> [route] [field] [origin] [arrive] [dest] [dep_time] [arr_time] <e>

```
field:1,1
city:3,3
city:6,6
number:8,8
origin:2,3:[city:3,3]
dest:4,6:[city:6,6]
loc:5,6:[city:6,6]
dep_time:7,9:[number:8,8]
arr_time:7,9:[number:8,8]
arrive:4,9:[dest:4,6],[arr_time:7,9]
route:1,9:[field:1,1],[origin:2,3],[dest:4,6],[dep_time:7,9]
route:1,9:[field:1,1],[origin:2,3],[dest:4,6],[arr_time:7,9]
route:1,9:[field:1,1],[origin:2,3],[arrive:4,9]
```

Figure 3
The Parsing Process

The parser proceeds left-to-right attempting to match each slot network starting with each word of the input, as in:

```
for (each word of input)
    for (each active slot)
            match_net ( slot, word )
```

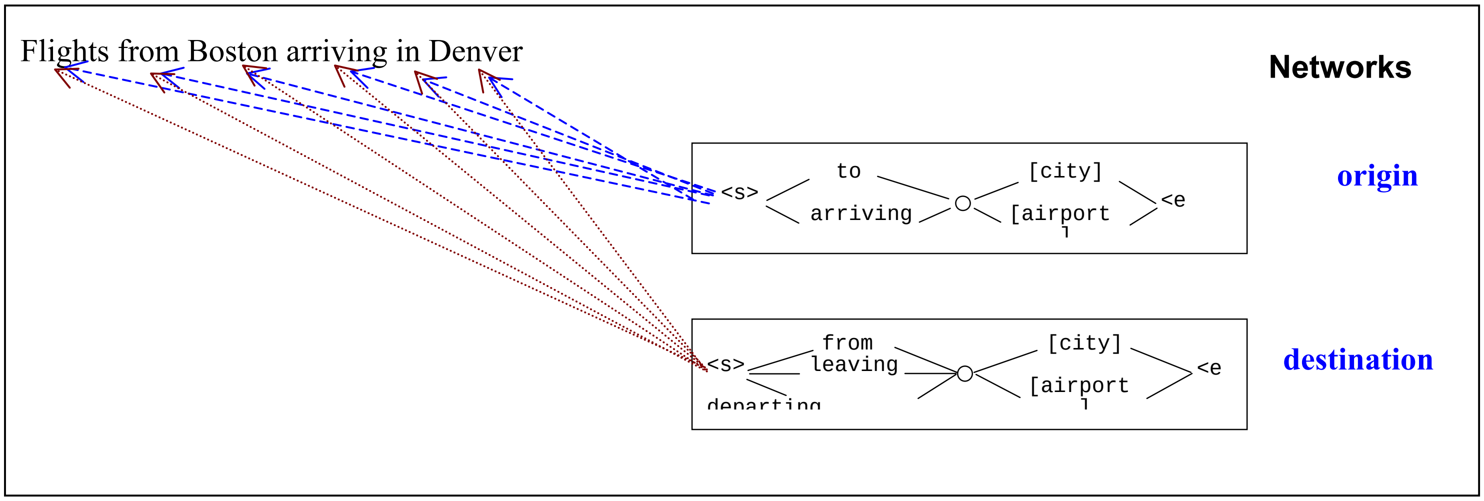


Figure 4
The Match Process

The function match_net is a recursive function that matches an RTN against a word string beginning at the specified word position. The function produces all matches for the network starting at the word position, and may have several different endpoints. The networks are not designed to parse full sentences, just sequences of words. The Recursive Transition Networks for the slots call other nets in the matching process. Each time a net match is attempted (all nets, not just slots), this is noted in the chart. All matched networks are added to the chart as they are found. Any time a net match is attempted, the chart is first checked to see if the match has been attempted before. When a slot match is found, it is added to the slot graph. Each sequence of slots in the slot graph is a path. The score for the path is the number of words accounted for by the sequence. Words are not skipped in matching a slot, but words can be skipped between the matched slots. The graph growing process prunes poor scoring paths, just as the acoustic search does. The pruning criteria are first, number of words accounted for and second, the degree of fragmentation of the sequence. If two paths cover the same portion of the input and one accounts for more words than the other, the less complete is pruned. If the two paths account for the same number of words, and one uses fewer slots than the other, the one with more slots is pruned. The resulting graph represents all of the sequences found that have a score equal to the best. The sequences of slots represented by the graph are then grouped into frames. This is done simply by assigning frame labels to the slots. Again in this grouping, less fragmented parses are preferred. This means that if two parses each have five slots, and one uses two frames and the other uses three, then the parse using two frames is preferred. The result is a graph of slots, each labeled with one or more frame labels. Each path through the graph, all scoring equally, is a parse. This mechanism naturally produces partial or fragmented parses. The dynamic programming search produces the most complete, least fragmented parse possible, given the grammar and the input.

The parser does not require sentence boundaries, and is able to process very long input with lo newline. It segments the input at natural breakpoints and performs garbage collection on its buffers. This allows it to parse entire reports as single utterances if necessary. The processing speed is generally linear with the length of the input.

How robust or constrained a system is depends on how the frames and grammars are structured. Using one frame with one slot will produce a standard CFG parser. This is efficient, but not very robust to unexpected input. At the other extreme, making each content word a separate slot will produce a key-word parser. Somewhere in between usually gives the best combination of robustness and precision.

Writing Semantic Grammars and Frames

The parser requires two input files, a frames file and a grammar file.

The *frames* file

This file specifies the frames to be used by the parser. A frame represents some basic type of action or object for the application. Slots in a frame represent information that is relevant to the action or object. Each slot name should have an associated set of grammar rules. The slot name will be the root of the corresponding semantic parse tree.

The syntax for a frame definition is:

```
# comment
FRAME: <frame name>
NETS:       slot names
          [<slot name>]
          [<slot name>]
          [<slot name>]
;
```

For example a frame for a Hotel query might be:

```
#   reserve hotel room
   FRAME: Hotel
   NETS:
      [hotel_request]
      [hotel_name]
      [hotel_period]
      [hotel_location]
      [Room_Type]
      [Arrive_Date]
      [want]
;
```

A frame definition is terminated by a ';' at the start of the line.
DO NOT FORGET TO TERMINATE THE FRAME WITH A SEMICOLON OR IT WON'T BE RECOGNIZED.

The Grammar file

Names of grammar files end with a ".gra" extension. These files contain the grammar rules. The grammars are context free rules that specify the word patterns corresponding to the token (network name).

The syntax for a grammar for a token is:

```
# optional comment
[token_name]
   (<pattern a>)
   (<pattern b>)
<Macro1>
   (<rewrite rule for Macro1>)
<Macro2>
   (<rewrite rule for Macro2)
;
```

The token name is enclosed in square brackets.
After this follow a set of re-write patterns, one per line each enclosed in parentheses, with leading white space on the line.
After the basic re-write patterns follow the non-terminal rewrites,
which have the same format.

Notation used in pattern specification:

- Lower case strings are terminals.
- Upper case strings are macros.
- Names enclosed in [] are non-terminals (calls to other token rules).
- Regular Expressions
  **item* indicates 0 or 1 repetitions of the item
  + indicates 1 or more repetitions
  +* indicates 0 or more repetitions (equivalent to a Kleene star)
- #include <filename> reads file at that point.

A macro has rewrite rules specified later in the same grammar rule. These cause a text substitution of all of the expansions for the macro, but don't cause a non-terminal token to appear in the parse. Macros allow for a simpler expression of the grammar without inserting unwanted tokens in the parse.

Example grammar rules for the tokens [hotel_request] and [want] are:

```
[hotel_request]
        (*[want] *a HOTEL)
HOTEL
        (hotel)
        (motel)
        (room)
        (accommodations)
        (place to stay)
;

[want]
        ( *I WANT)
I
        (i)
        (we)
WANT
        (want)
        (need)
        (would like)
;
```

The parse for *I would like a hotel room* would be:

```
[hotel_request]( [want]( i would like) a hotel room)
```

Compiling Grammars

The grammar is compiled by invoking the script *compile*, in the Grammar directory, which in turn calls the routine *compile_grammar*. A sample *compile* script is shown in Figure 5. The grammar can be in a single file or spread across multiple files. If the grammar is in multiple files, as indicated by the variable *SingleFile* set to 0, the *compile* script concatenates them to a single file before passing it to *compile_grammar*. It also appends any requested grammars from Phoenix/Grammars to the file before invoking *compile_grammar*. The variable *LIBS* is set to the set of grammars to be loaded from the common library. The following is the script for compiling the grammar for the tutorial example. It uses two grammars in the local directory, *Places.gra* and *Schedule.gra*, and it loads grammars *date_time.gra*, *number.gra*, and *next.gra* from the common library. The individual grammars are concatenated into the file *EX.gra* which is then compiled into the file *EX.net*. The compile script creates the file *nets* before compiling the grammar. This file lists the names of nets in the grammar. It is read by *compile_grammar* and used to assign numbers to nets. *compile_grammar* also creates the file *base.dic* in the local directory. This is the dictionary, a text file containing the words used by the grammar with a number assigned to each. Each line in the file contains one word and the number assigned to it. The word numbers are not sequential, they are assigned by hash code. The word numbers are used by the parser.

```
# compile grammars in current directory
set DIR=`pwd`
# put compiled nets in file EX.net
set TASK="EX"

# set PHOENIX to point to root of Phoenix system
set PHOENIX="../../../Phoenix"

# set LIBS to .gra files to load from $PHOENIX/Grammars
set LIBS="date_time.gra number.gra next.gra"

# if SingleFile == 1, then all grammar rules are in file $TASK.gra
# if SingleFile == 0, then compiles all files in dir with extension .gra
@ SingleFile = 0

# if separate files, pack into single file, $TASK.gra
if( $SingleFile == 0 ) then
   cat *.gra > xxx
   mv xxx $TASK.gra
endif

# append lib grammars to file
cd $PHOENIX/Grammars
cat $LIBS >> $DIR/$TASK.gra
cd $DIR

# create list of nets to be compiled
cat $TASK.gra | $PHOENIX/Scripts/mk_nets.perl > nets

# compile grammar, output messages to file log
echo "compiling grammar"
$PHOENIX/ParserLib/compile_grammar -f $TASK > log
grep WARN log

# flag leaf nodes for extracts function
echo "flagging leaf nodes"
$PHOENIX/ParserLib/concept_leaf -grammar $TASK.net
```

Figure 5
compile script

.net file - compiled grammar file
These are ascii files representing Recursive Transition Networks and have the following format:
The first line gives the number of nets that were compiled
Number of Nets= 378

Then come the compiled nets. For each network:
The first line gives: net name, net number, number of nodes in the net, concept leaf flag
Then the nodes are listed, each followed by the arcs out of it.
A node entry in the file has the following format:
node number, number of arcs out of node, final flag: 0= not a final node 1= a final node
for each net, nodes are numbered sequentially, with the start node being 0.
The entries for arcs out of a node follow the entry for the node.
Arc entries have the following format:
word_number net_number to_node
An arc may be a:

- word arc, with word word_num and net_number=0
- null arc, indicated by word_num=0 and net_number=0
- call arc, indicated by net_num >0 (word_num is ignored in this case)

```
[Airline] 27 5 0
0  5 0
            0    0    2
            20294    0    3
            372    372    4
            344    344    2
            30    30    2
1  0 1
2  1 0
            72    72    1
3  3 0
            18753    0    1
            18762    0    1
            16707    0    1
4  3 0
            18753    0    1
            18762    0    1
            16707    0    1
```

Data Structures

The Grammar structure:

Phoenix is capable of using multiple grammars. The function read_grammar reads a compiled grammar from a .net file and returns a pointer to an associated gram structure.

```
struct gram *read_grammar(dir, dict_file, grammar_file, frames_file);

typedef struct gram
{
   FrameDef     *frame_def;         /* pointers to frame definition structures */
   char        **frame_name;        /* pointers to frame names */
   int           num_frames;        /* number of frames read in */
   char        **labels;            /* pointers to names of nets */
   Gnode       **Nets;              /* pointers to heads of compiled nets */
   int           num_nets;          /* number of nets read in */
   char        **wrds;              /* pointers to strings for words */
   int           num_words;         /* number of words in lexicon */
   int          *node_counts;       / * number of nodes in each net */
   char         *leaf;              /* concept leaf flags */
   char         *sym_buf;           /* strings for words and names */
} Gram;
```

As many separate grammars as desired can be read in, and the pointers to the corresponding gram structures stored. When the parse function is called, it is passed the word string to parse and a pointer to the gram structure to use for the parse:
parse(word_string, gram).

Set of Active Slots:

In addition to having control of which grammar to use in the parse, the developer also has dynamic control of the set of slots to use when invoking each parse. The global variables *cur_nets* (int *cur_nets) and *num_active* (int num_active) are used to implement this capability. *cur_nets* points to the set of slots (the net numbers) to be used in the parse and *num_active* specifies the number of elements in the array. The main match loop of the parse looks like:

```
for( word_position=1; word_position < script_len; word_position++ ) {
    for( slot_number= 0; slot_number < num_active; slot_number++ ) {
        match_net( cur_nets[slot_number], word_position, gram)
```

If *cur_nets* is null, the parser will create the set of nets used by frames in the grammar, point *cur_nets* at it and set *num_active* accordingly. The system will therefore initialize to this the first time *parse()* is called. In order to use a subset of all slots in the parse, generate an array of net numbers to used, point *cur_nets* at the array and set *num_active* to the number elements in the array. In order to reset back to the set of all slots in a grammar, just set *cur_nets* or *num_active* to null.

The Chart

The chart is a data structure used to record all net matches as they are found. This is done so that the match doesn't have to be repeated if it is called from some other point in the grammar. In some cases, this saves considerable computation. The chart is a triangular matrix that is indexed by the start word and end word of the matched sequence of words (Figure 5).

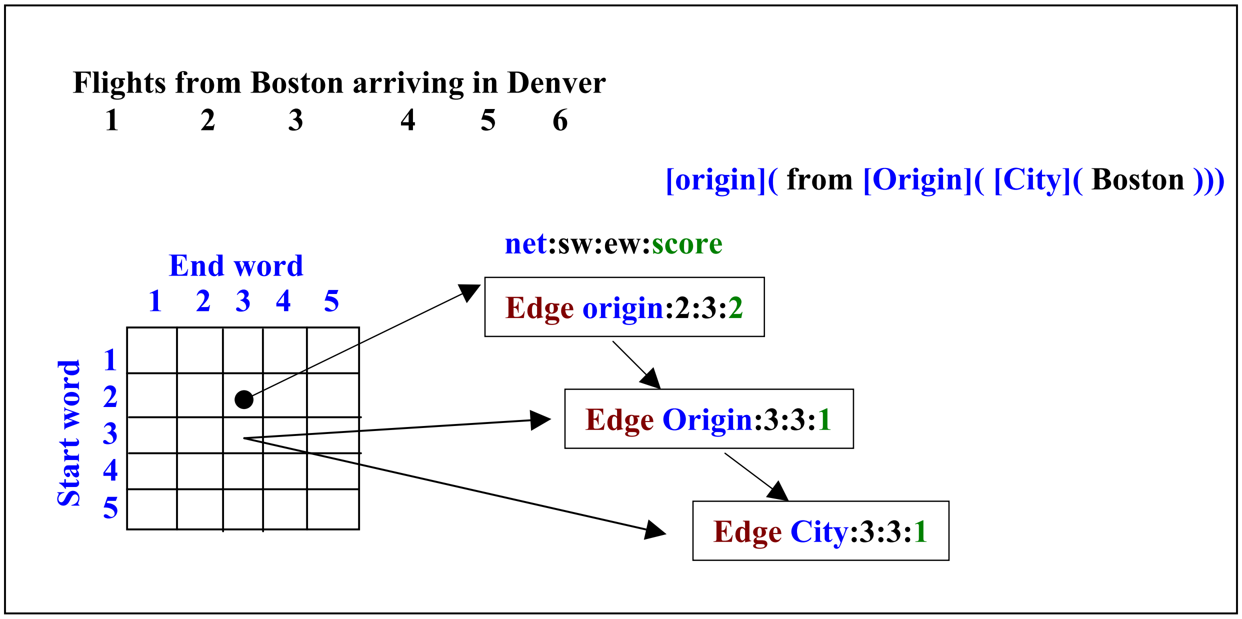


Figure 5
The Chart

Since it would be a sparse matrix, the chart is not actually implemented as a matrix, but as a linked list with the start word as the major key and the end word as the minor key.

```
EdgeLink        **chart;

/* structure for linking edges into chart */
typedef struct edge_link {
          int                   sw;
          struct edge_link      *link;
          Edge                  *edge;
}

typedef struct edge {
          short   netid, sw, ew, score;   /* net number, start word, end word, score */
          char    nchld;                  /* number of child edges */
          struct edge      **chld,        /* edge called by this edge */
                           *link;         / link to next edge in list */
} Edge;
```

Parse Data Structure

When the final parse structures are created, they are put into a buffer pointed to by the variable *parses*. A parse is a sequence of SeqNodes, which represent edges each with a frame id. The SeqNode structures contain a frame id and a pointer to an edge in the chart. A sequence of edges with the same frame id are all in the same instance of a frame. The nodes are written to the buffer in order, with the first slot of a new parse appended after the last slot of the previous parse. All parses for an utterance will have the same number of slots filled, since any more fragmented ones would have been pruned. The number of slots in each parse is in the global variable *n_slots*. The number of alternative parses is in *num_parses*.

```
SeqNode    **parses;

typedef struct seq_node {
        Edge   *edge;
        Fid    *frame_id;
        short  n_frames; /* frame count for path */
} SeqNode;
```

Printing Parses

The routine *print_parses()* is used to write parses to a text buffer. The parameters are:

- the number of the parse to be printed (staring at 0)
- a pointer to the text buffer to be written to
- the extracted form flag (0= parsed form, 1= extracted form)
- a pointer to the grammar structute to use

```
void print_parse(int parse_num, char *out_str, int extract, Gram *gram)
{
    int          frame;
    SeqNode   *fptr;
    char       *out_ptr;

    fptr= parses[parse_num];
    for(j=0; j < n_slots; j++, fptr++) {
         if( fptr->n_act != frame )  frame= fptr->n_act;   /* if starting a new frame */

        sprintf(out_ptr, "%s:", gram->frame_name[frame]); /* print frame name and colon */
        if( extract )
            out_ptr= print_extracts( fptr->edge, gram, out_ptr, 0, gram->frame_name[frame]);
        else
            out_ptr= print_edge( fptr->edge, gram, out_ptr);    /* print slot edge */

    }
}
```

As and example of calling *print_parse()*, the *parse_text* routine uses the following for loop to print out the parse buffer:

```
/* print all parses to parse buffer */
for(i= 0; i < num_parses; i++ ) {
        sprintf(out_ptr, "PARSE_%d:\n", i);
        out_ptr += strlen(out_ptr);

        print_parse(i, out_ptr, extract, gram);
        out_ptr += strlen(out_ptr);

        sprintf(out_ptr, "END_PARSE\n");
        out_ptr += strlen(out_ptr);
}
```

Extracted Form

The parser provides a mechanism for printing output more directly like that which is required by a dialogue manager. It will print only the tokens and strings that are to be extracted. The format for extracted output is:

```
<frame_name>:<Node_Name>.[<Node_Name>.[<Node_Name>.]] [<value>]
```

For example: *Where is American Beauty playing?* Would generate parsed and extracted forms:

```
movie_info:[movie_info] (
    [Display_Movie] ( [_theatre__name] ( where ) ) [is] (is)
    [Movie] ( AMERICAN BEAUTY ) [showing] ( playing ) )
```

$$\Downarrow$$

```
movie_info:[Display_Movie].theatre_name
movie_info:[Movie].AMERICAN BEAUTY
```

In order to use this mechanism, the developer must observe a convention when writing grammars. Nets that begin with an uppercase letter will appear in the extracted output token tree, and the leaves of those nets will be the extracted values. Nets that begin with a lower case letter will not appear in the extracted output. Word strings that have only lower case nets in their parse will not appear in the output.

Pre-Terminals

A second feature of extracted output are pre-terminals. These are nets whose name begins with an underscore (_) character. For these nets, the name of the net rather than the word string is used as the value in the extracted output. See *_theatre_name* in the example. This is a convenient way to put values in canonical form. For example, there could be many ways of saying the word *yes*, but you might need to pass the exact string *yes* to the database. If you define a net *[_yes*] that specifies all the ways of saying *yes*, then the string *yes*, rather than the original word string (ex: *certainly*) will appear in the extracted output. This also works for mapping strings like *Philly* to *Philadelphia*, etc.

As an example, the following rules:

```
[Answer]
        ([_yes])
        ([_no])
;

[_yes]
        (yes)
        (fine)
        (sure)
        (that's good)
        (sounds good *to *me)
;

[_no]
        (no *way)
        (*i don't think so)
;
```

would cause *sounds good to me* to be parsed as:

[Answer] ([_yes] (sounds good to me))

and extracted as:

[Answer].yes

and *I don't think so* to be parsed as:

[Answer]([_no] (I don't think so))

and extracted as:

[Answer].no

Running the Parser

Command-Line parameters

Required parameters:

-dir <grammar directory>
Specifies the directory which contains all grammar files.

-grammar <filename>
Name of the .net file containing the grammar to be used.

Flags:

-verbose <number>
Number can have values from 0-6. As number increases, progressively more trace information is printed out. 0 prints out nothing. 1 prints out just parses, this is the default. 2 prints out the input string and then the parses. 3 – 6 print out debugging information.

-extract 0/1
0 is default and prints full bracketed parses
1 prints extracted forms

-PROFILE 0/1
1 Turns on profiling. This gives information on parse times and buffer sizes. This can be used to optimize the sizes of the internal structures allocated. Default is 0.

-IGNORE_OOV 0/1
0 causes out of vocabulary words to prevent a rule match
1 causes out of vocabulary words to be ignored and not prevent matches. Default is 1.

-ALL_PARSES 0/1
0 causes the system to print out only one parse.
1 prints out all parses in the final parse forest. 1 is the default.

-MAX_PARSES <number>
Print out at most <number> parses. Default is 10.

-USE_HISTORY 0/1
0 Do not inherit context from previous sentence.
1 Inherit final context from previous sentence for fragmentation score.
0 is the default.

-BIGRAM_PRUNE 0/1
1 causes more aggressive pruning in the parse search. The dynamic programming search for the best parse only keeps one path for each *end_word/frame_id/ slot* triplet. This amounts to bigram level pruning with a Viterbi search.
Default is 0.

- MAX_PATHS 0/1
Set to 0, this parameter has no effect. If >0, it specifies the maximum number of paths to be expanded during a match. This is for debugging in case a "runaway" rule is causing large amounts of output.
Default is 0.

Optional Files:

-function_wrd_file <filename>
File containing stop word list. These are function words that should not be counted when scoring the goodness of a parse

-config_file <filename>
Read command line parameters from the specified file.

Change Default Symbols:

-start_sym <ascii string> - Set the start of utterance symbol to string. Default is <s>

-end_sym <ascii string> - Set the end of utterance symbol to string. Default is </s>

Buffer Sizes:

The parser mallocs space for buffers. Each of these has a default size. If during execution a buffer runs out of space, the parser will print an error message telling which buffer is out of space and what the current setting is. The command line parameter can be used to increase or decrease the buffer size. These all have default values. The PROFILE parameter can be used to report how much space is actually used during parses if desired to optimize the buffer sizes.

```
EdgeBufSize=    1000,   /* buffer for edge structures */
ChartBufSize=   40000,  /* chart structures */
PeBufSize=      2000,   /* number of Val slots for trees  */
InputBufSize=   1000,   /* max words in line of input */
StringBufSize=  50000,  /* max words in line of input */
SlotSeqLen=     200,    /* max number of slots in a sequence */
FrameBufSize=   500,    /* buffer for frame nodes */
SymBufSize=     50000,  /* buffer size to hold char strings */
ParseBufSize=   200,    /* buffer for parses */
SeqBufSize=     500,    /* buffer for sequence nodes */
PriBufSize=     2000,   /* buffer for sequence nodes */
FidBufSize=     1000;   /* buffer for frame ids */
```

Stand Alone Interface

A stand alone interface, *parse_text*, is provided. *parse_text* accepts lines of input from stdin and prints output to stdout. Type *qui*t to exit. A script for running *parse_tex*t, *run_parse*, is in the *Example* directory**.** It looks like:

```
        $PHOENIX/ParserLib/parse_text –config config/parse_config
```

where *parse_config* is

```
verbose:1
extract:1
dir: Grammar
grammar: EX.net
```

The parse_text routine illustrates the basic setup needed to use the parse functions:

```
int main(argc, argv)
{
        /* set command line or config file parms */
        config(argc, argv);

        /* read grammar, initialize parser, malloc space, etc */
        init_parse(dir, dict_file, grammar_file, frames_file, priority_file);

        /* for each utterance */
        while( fgets(line, LINE_LEN-1, fp)  ) {

                /* assign word strings to slots in frames */
                parse(line, gram);

                /* print parses to buffer */
                for(i= 0; i < num_parses; i++ )
                        print_parse(i, out_ptr, extract, gram);

                 /* clear parser temps */
                reset(num_nets);
        }
}
```

Tutorial Example

The *Example* directory contains a tutorial example.

First, cd to the *Example/Grammar* directory and type *compile* to compile the grammar. All files in this directory, including the compiled *EX.net* file are acsii. There are two source grammar files (*.gra*) in the directory, *Places.gra* and *Schedule.gra*. The compile script creates the *EX.gra* file (after first saving the old one to *EX.gra.old*) from these two files and the ones specified from the *$PHOENIX/Grammars* directory. It then compiles the grammar into the *EX.net* file. The script also creates the *nets* and *base.dic* files. The files *airports*, *cities* and *states* are included by the *Places.gra* grammar.

Then cd to the *Example* directory(..). The file *run_parse* is a script to run *parse_text* with the command line parameters specified in *config*. The file *input* contains some sample sentences that the example grammar should parse. To process the sentences type *run_parse < input*. Edit the config file and change the extract parameter to be 1, and run the parser on the input again.