Introduction

This quagent (nicknamed "Gabber") communicates in a simple natural language. It relies on a table-driven parsing of an LL(1) grammar that is specifically designed to mimic a command-and-query structure. In other words, because quagents can do so little (they can walk, pick up, drop, run, etc.), the natural language understanding is little more than an interpreter to and from the quagent protocol. While communication with a quagent is more concisely and unambiguously facilitated through use of a simple protocol, that does not mean that the natural language interface is without advantages. The interpreter presented here allows for task queuing, interlaced queries and orders, and a robust (and easily extensible) lexicon of synonyms. Some of the strong and weak points of this interface will be addressed below.

While this report will showcase the results of the particular grammar supplied with this code, it also emphasizes the easy extensibility of this system to a larger or more complicated problem domain. In general, "how can this be applied to a larger problem" seems to be in the general spirit of this project, so hopefully any digressions will be forgiven. For this reason, the report occasionally calls attention to developer-side features (such as the easy of extensibility of the grammar) as well as user-side features.

General Structure

For instructions on the code layout or running the program, please consult the readme in the code directory, attached to this submission. The following section describes the overall program structure.

As stated above, the program relies on a table-driven parsing of an LL(1) grammar. The majority of the code in the Gabber.java file is oriented around building the driver tables and loading needed information from the grammar definition. Lines of user input are then translated into streams of tokens by a scanner. These streams are parsed into trees by a parser, and then finally examined by a semantic analyser. Only the analysing logic is "hard-coded" into Gabber.java - the scanner and parser is entirely dependant on grammar.txt.

Loading the Grammar

There's an ugly block of static{ } code that loads and interprets the grammar file when Java loads the Gabber class. The format of the file is fairly straightforward and line-break delimited, so the interpretation isn't bad. The grammar.txt file itself has merely a list of terminals, productions, first sets, and follow sets. If the grammar is not LL(1), the code will complain as helpfully as it can (by specifying where common prefixes occur). It will not, however, stop left recursion, so any expansions to the grammar must be careful to avoid causing it. An example error message at this stage is, "Error! Grammar is not LL(1): common prefix for productions of nonterminal 1 when nullability is considered. (8: want to use production 2, but already using 0)".

Scanning

The scanner loads terminal definitions from grammar.txt and places them into a hash map with the terminal "words" as keys, and the integer value of the terminal as values. Tokens are split by spaces, though there are some exceptions that complicate the scanning code but add a great amount of expressive power to the grammar by simplifying complicated phrases that often appear together into single tokens. For example, there is a "would you" token as well as a "tell me" token - both of these have to look across multiple words in the input stream to build one token, but lower the overall complexity of the grammar as well as provide more helpful error messages (after all, the quagent does not understand "tell" in general, so it makes sense

for it to protest that it doesn't know that word except in a few narrow cases). Multiple-word tokens are specified in the grammar.txt file simply as multiple words enclosed in quotation marks.

The scanner also possesses the power to handle so-called "postfix tokens". A postfix token is a token that can appear after a prior legal token without the usually-required space. While these tokens only include periods, commas, exclamation points, and question marks in the current version of the grammar, they could conceivably be used to handle hyphenated words or complex words with suffix-based semantics (plurals or past tenses). The current version of the grammar merely uses postfix tokens to properly represent punctuation (it's worth noting that while the quagent doesn't react differently between an order ending in a period or an exclamation point (or no punctuation at all), it can tell the difference at any stage of the command interpretation).

The scanner is also very good at synonyms. These are simply words that are mapped to token values that already have some semantic meaning assigned to them. For example, "get" and "pick up" are semantically equivalent in the grammar, and to reduce the number of branches and the size of the terminal set, the scanner tokenizes them to the same token. Expanding the list of synonyms is as simple as adding new lines in the grammar.txt file - no recompilation, production adjustment, or first/follow set updating is necessary.

There is another special token that the scanner handles: the "::SCALAR::" token, which reads in any number that java can format as a Double. This offers just a little extra flexibility in number specification: both "22.2", and "2.22e1" are legal scalars, and are interpreted correctly.

The scanner also encodes the original value of the text matched along with the token, so that later errors can offer some more detail on where the error occurred (for the scalar token, it instead encodes the numeric value matched). Errors during the scanning phase always are of the form, "Sorry, I don't know the word 'WORD'. Guess I'm just too dumb." These are typically pretty helpful, since they point out precisely what piece of the sentence is not understood.

Parsing

The parser relies entirely on the table filled out by the static analysis of the grammar.txt file. It is a standard table-driven parser with no peculiar tricks: it simply pushes productions onto a stack and iteratively pops nodes off the stack to expand them based on the predict sets of the nonterminals (which are a combination of the first and follow sets). As it does so, it generates a parse tree that contains encoding to aid in semantic analysis. More information on table-driven parsers is available on-line.

Errors during the parsing step can be helpful or unhelpful, depending on the input. Typically, for small lines of input, they're very helpful, since they identify exactly where the problem is taking place. However, on long lines of input, such as the following: "Would you run 5 yards southwest, rotate 1 radian, walk 3.212e3 inches north, spin 1.88 radians, tell me where you are, and then drop the gold?", a rather mysterious "You confused me at u",", sorry." error is returned, and it's made all the worse because to most casual speakers it is not apparent why the commas present a problem!

What the quagent is trying to communicate is that it saw a comma token that was taken as a postfix token (hence the u"X" notation) when it didn't think one was legal. A closer inspection of the instruction with this in mind quickly reveals the cause of the problem: the user never said which way to turn! This particular example is explored more in Appendix B.

Semantic Analysis

The semantic analysis step takes place in two phases, through two functions: understand() and

execute(). Understand simply strips away the high-level sentence organization, and sets up a queue of orders or queries (the two fundamental things that can be issued to a quagent, based on the quagent protocol). These commands are then execute()-ed in sequence. Each command blocks the other commands until it is complete, which conveys the semantic meaning of a string of commands. For example, if the user types, "Please walk 20 feet north and then tell me where you are.", the understand() function is responsible for splitting the parse tree into two trees, one of which encodes the "walk 20 feet north" command, and the other that encodes "tell me where you are", which is expected to chronologically follow. This is a very straightforward process because the grammar has been organized to facilitate such an easy separation (see the Grammar section).

The execute() function then walks the tree in a fixed pattern much like a recursive descent analyser does (except the only recursive part of execute() is when a query is actually saying, "will you ORDER", or an order is saying, "tell me QUERY"). If the grammar is altered, the execute() function will have to be altered as well to provide new semantic rules for interpreting the grammar into an action sequence.

Grammar

The full specification of the grammar is reproduced in Appendix B. Here, I will only discuss a few interesting elements of it.

The grammar submitted with this assignment was carefully designed so that all commands and queries in a single input string are entirely represented in disjoint subtrees of the large tree. This makes separating them easy - the understand() function simply finds the root nodes of these subtrees and pretends they're completely new action trees. This representation also allows the execute() function to exist with simple fixed-depth exploration, since every command it evaluates is guaranteed to begin at either a query or order node (each of which has a known and fixed substructure).

The grammar is also a little more general than necessary: for example, the string "Turn 10 degrees left and and and and then, and then then, and and then, walk 20 feet." will parse just as easily as "Turn 10 degrees left and walk 20 feet." In fact, after understand() is through with these strings, they're identical. Internally, both "and", "then", and u"," are turned into "," tokens, and the QCOMMA -> "," QCOMMA production allows an arbitrarily large number of commas to appear in a row. This is a handy short-cut, since it's almost always legal (or even natural) to just say "and" or "then" when an "and then" is really properly required.

The grammar allows for both presence and lack of punctuation, both of which are fairly common modes of text-based communication - some users will likely prefer to punctuate fully, while others may choose to not spend the effort. The one exception to this rule is that queries must end in a question mark. The scanner is case-insensitive, so capitalization information is actually lost during tokenizing. Capitalization might be handy for some semantic analysis, such as disambiguating between "Penny" and "a penny", but this grammar does not pay attention to proper nouns. Even when capitalization might be useful, it is unreliable in informal word (especially in text messaging interfaces, which is essentially what this project mimics).

Appendix A: Session Log

The following is a demonstrative sample of interaction with the quagent. The following text has been altered from its original version. It has been formatted to fit your screen. (In other words, indentation has been inserted on machine responses.) User-typed messages are boldfaced.

```
Turn 120 degrees clockwise.
      All this turning makes me dizzy.
Walk 120 inches.
      Wok wok wok.
Please walk 20 feet north.
      Wok wok wok.
Pick up the gold.
      There's no gold around here!
Walk 1.5 yards south then turn 90 degrees left and walk 1 yard, then pick up
the gold.
      Wok wok wok.
      All this turning makes me dizzy.
      Wok wok wok.
      Lift with your legs, not with your back...
Where are you?
      I'm at -42.7996x, and -12.7999y, now. I'm facing 360.0 degrees.
Would you please expound upon the nature of the dichotomy of creation and
destruction?
      Sorry, I don't know the word 'expound'. Guess I'm just too dumb.
Would you run 5 yards southwest, rotate 1 radian, walk 3.212e3 inches north,
spin 1.88 radians, tell me where you are, and then drop the gold?
      You confused me at u",", sorry.
      Could you use a better sentence structure, maybe?
Would you run 5 yards southwest, rotate 1 radian left, walk 3.212e3 inches
north, spin 1.88 radians right, tell me where you are, and then drop the
gold?
      I hate running, but I'll do it for you.
      All this turning makes me dizzy.
      Wok wok wok.
      All this turning makes me dizzy.
      I'm at -145.693x, and 234.706y, now. I'm facing 342.284 degrees.
      Whew! Glad I can put that down.
Next, tell me your location, pick up the gold, walk 1 foot forward, then run
5 yards right.
      I'm at -145.693x, and 234.706y, now. I'm facing 342.284 degrees.
      Wok wok wok.
      Lift with your legs, not with your back...
      I hate running, but I'll do it for you.
Run 2 miles!
      I hate running, but I'll do it for you.
Run 2 miles backward!
      I hate running, but I'll do it for you.
run 2 miles right
      I hate running, but I'll do it for you.
die
      Die?! But there was so much more I wanted to do...
      *choke*
      Ack! Gkk!
      Bleeeuuuurgggh...
      ...
```

Appendix B: grammar.txt

The following is the text of grammar.txt, which can also be found among the submitted files. It has been reformatted to fit the width of this document, and avoid text-wrapping.

```
# This file defines the "semantic grammar" that the agent uses: this is
# an LL(1) grammar designed so that the parse structure mimics the
# semantic structure.
# Lines that begin with "#" are comments. Blank lines are not currently
# allowed.
# The scanner tokenizes words by spaces. However, there are some
# exceptions:
#     Punctuation marks (.,!?) need not have a space prior to them:
# they are tokens unless they appear within a token (e.g. "next,").
#     Some tokens are multiple words. In cases where "X Y" is a token,
# but "X" or "Y" is as well, whichever appears higher in the following
# list will be the assignment.
#         That is, this list is in order of descending precedence.
#
#
# Terminals:
@TCOUNT:  46
@NTCOUNT: 16
    "yes"               = 0
    "no"                = 1
    "next"              = 2
    u"."                = 3
    u"!"                = 4
    u"?"                = 5
    u","                = 6
    "please"            = 7
    "tell me"           = 8
    "would you"         = 9
    "a"                 =10
    "get"               =11
    "drop"              =12
    "walk"              =13
    "run"               =14
    "the"               =15
    "turn"              =16
    "feet"              =17
    "yards"             =18
    "miles"             =19
    "forward"           =20
    "backward"          =21
    "left"              =22
    "right"             =23
    "north"             =24
    "south"             =25
    "east"              =26
    "west"              =27
    "northwest"         =28
    "northeast"         =29
    "southwest"         =30
    "southeast"         =31
    "degrees"           =32
```

```
    "radians"          =33
    "clockwise"        =34
    "counterclockwise" =35
    ::SCALAR::         =36
    "inches"           =37
    "tofu"             =38
    "battery"          =39
    "box"              =40
    "gold"             =41
    "data"             =42
    "kryptonite"       =43
    "head"             =44
    "where you are"    =45
# Synonyms:
    "an"               =10
    "pick up"          =11
    "grab"             =11
    "put down"         =12
    "set down"         =12
    "go"               =13
    "and then"         = 6
    "and"              = 6
    "then"             = 6
    "rotate"           =16
    "spin"             =16
    "back"             =21
    "degree"           =32
    "radian"           =33
    "inch"             =37
    "foot"             =17
    "yard"             =18
    "mile"             =19
    "your position"    =45
    "where are you"    =45
    "your location"    =45
@END_TERMINALS_START_NONTERMINALS
# Productions (numbers are nonterminals, _numbers are terminals, |s are
# ORs, -1 is epsilon): 11 and 12 are unfinished.
  0 : 4 1 3 | 11 _5
  1 : 5 15 1 | 2 15 1 | _6 15 1
  2 : _0 | _1
  3 : _3 | _4
  4 : _2 15 | _7
  5 : _8 11 | _11 6 | _12 6 | _13 7 8 | _14 7 8 | _16 9 10
  6 : _10 12 | _15 12
  7 : _36 13
  8 : _20 | _21 | _22 | _23 | _24 | _25 | _26 | _27 | _28 | _29 | _30 | _31
  9 : _36 14
 10 : _22 | _23 | _34 | _35
 11 : _45 | _9 5 15 1
 12 : _38 | _39 | _40 | _41 | _42 | _43 | _44
 13 : _17 | _18 | _19 | _37
 14 : _32 | _33
 15 : _6 15
@NULLABLE
  1
  3
```

```
  4
  8
 11
 12
 15
@FIRSTSETS
  0 : 0 1 2 3 4 6 7 8 9 11 12 13 14 16
  1 : 0 1 8 11 12 13 14 16
  2 : 0 1
  3 : 3 4
  4 : 2 7
  5 : 8 11 12 13 14 16
  6 : 10 15
  7 : 36
  8 : 20 21 22 23 24 25 26 27 28 29 30 31
  9 : 36
 10 : 22 23 34 35
 11 : 45 9
 12 : 38 39 40 41 42 43 44
 13 : 17 18 19 37
 14 : 32 33
 15 : 6
@FOLLOWSETS
  1 : 3 4
  4 : 0 1 3 4 8 11 12 13 14 16
  8 : 0 1 3 4 6 8 11 12 13 14 16
 15 : 3 4 8 11 12 13 14 16
#
#
#
#
# The following is the original grammar I designed (some slight deviations
# might appear in the grammar above):
#
# INPUT     (0)   ->    PREFIX STATEMENT PUNC
#                 ->    QUERY?
# STATEMENT (1)   ->    ORDER QCOMMA STATEMENT
#                 ->    RESPONSE QCOMMA STATEMENT
#                 ->    {null}
# RESPONSE (2)    ->    yes
#                 ->    no
# PUNC (3)        ->    !
#                 ->    .
# PREFIX (4)      ->    next QCOMMA
#                 ->    please
#                 ->    {null}
# ORDER (5)       ->    tell me QUERY
#                 ->    get OBJECT
#                 ->    drop OBJECT
#                 ->    walk DISTANCE DIRECTION
#                 ->    run DISTANCE DIRECTION
#                 ->    turn ANGLE CLOCK
# OBJECT (6)      ->    the ENTITY
#                 ->    a ENTITY
# DISTANCE (7)    ->    [num] DISTANCEUNIT
# DIRECTION (8)   ->    forward
#                 ->    backward
```

```
#                 ->    back
#                 ->    left
#                 ->    right
#                 ->    north
#                 ->    south
#                 ->    east
#                 ->    west
#                 ->    northeast
#                 ->    northwest
#                 ->    southeast
#                 ->    southwest
# ANGLE (9)       ->    [num] ANGLEUNIT
# CLOCK (10)      ->    left
#                 ->    right
#                 ->    clockwise
#                 ->    counterclockwise
# QUERY (11)      ->    would you ORDER QCOMMA STATEMENT
#                 ->    where are you
# DISTANCEUNIT    ->    feet
#                 ->    meters
#                 ->    miles
# ANGLEUNIT       ->    degrees
#                 ->    radians
# QCOMMA (15)     ->    , QCOMMA
#                 ->    {null}
```