# NATHACK: a Natural Language Interface for Nethack

Cassia Martin, David Molnar, Dev Purkayastha, Noam Zeilberger

January 9, 2003

## 1 Introduction

Nethack is a popular "rogue-like" (Waijers, 2003) adventure game. Previous work on rogue-like games has focused on developing agents that can play the game at a high level of skill, such as the ROG-O-MATIC Belligerent Expert System (Mauldin et al., 1984). This paper addresses a different direction of research; we describe NATHACK, an English-language interface for Nethack.

The original user interface for Nethack is based on direct manipulation. As in most other rogue-like games, information about the world is represented through the use of ASCII symbols on screen. Figure 1 shows a screenshot from the game. The player directly manipulates these symbols by pressing particular keys. For example, in Nethack the player is represented by an @ sign; pressing the key for 'left' moves that sign one space left on the screen. To kill a monster, the player must enter a sequence of keystrokes such as "jjlllyFlFl". That particular example moves the @ sign on the screen to the sign representing the monster and then attacks twice to the right. Our system allows the user to issue that same command with more intuitive English sentence "move to the monster and kill it."

Previous work combining natural language interfaces and games has mostly focused on text-based adventure games or "interactive fiction." Popularized by Infocom in the early 80s, the hallmark of interactive fiction is that both the user's commands and the description of the world use natural language. One of the first such games, ADVENTURE, recognized only sentences of the form "verb noun" and matched words only on their first four characters (Crowther et al., 2001). Recent examples of interactive fiction provide more robust parsing and augment the game world with a logic for reasoning about user commands (Koller et al., 2002).

```
Idefix misses the jackal. Idefix bites the jackal. The jackal is killed!

     ----------------
     |..............|
     |..............|
     |..........%...|
     |.........@f...|                ------
     --------|----- -                |....-#######`
            ##                       |....|
            # (                      |.>..|       -------
              #                     #-....-#######-.....|
              #         ------    ###|....|       |.....|    ------
             -|-+--   ##.....|    #  ------       |.....|    |....|
             |....|   # |.<...##################`|......##  |....|
             |....-[### |..{.|#                  |.....| #  |....|
             |....|     |.....###################-.....| ###|....|
             |....|     |..!.|#                  -------   #.....|
             ------     ------                             ------
Chomsky the Candidate      St:18/01 Dx:14 Co:11 In:21 Wi:20 Ch:15 Lawful
Dlvl:1  $:58 HP:12(14) Pw:5(5) AC:4  Exp:1
```

Figure 1: Screenshot from Nethack

Rogue-like games do not use natural language, either to present their world or to communicate with the user beyond status messages. We are not aware of previous attempts to build a natural language interface to rogue-like games. Given how well the direct manipulation interface has served rogue-like game for over fifteen years, a natural language interface may seem unnecessary. When we look closer, several basic features of natural language interfaces are applicable to the Nethack domain, and it certainly provides a more intuitive command system for new users.

Marilyn Walker lists several features of natural language interfaces (Walker, 1989). The list below is presented along with sample sentences for a Nethack command interface.

- Definite Description
  Objects can be referred to by their qualities. For example, the sentence "Kill the jackal" picks out the object which is a jackal. The sentence "Kill the blue jelly" goes further and adds an adjective.

- Quantification
  Natural language allows quantifying over objects, such as in the sentence "Kill all the newts." This quantification is difficult in a direct manipulation system.

- Discourse Reference
  The sequence of commands to the game can be thought of as a discourse. Then the sentence "Where did I drop the Amulet of Yendor?" is understood as referring back to previous commands and has a well-defined answer.

- Temporal Specification
  "Kill the jelly two turns from now." "Rest until healed."

- Coordination
  Actions can be sequenced, as in the sentence "Move left and then kill the monster." We will see later that this leads to interesting questions of reference.

- Negation
  Quantification in natural language can be extended by negation, as in "Pick up all items that are not cursed." Again, this is difficult to achieve with a direct manipulation interface.

- Comparatives
  A comparative term discriminates between objects. The sentences

"Run to the nearest door" and "Kill the strongest monster" show how natural language comparatives can find natural uses in Nethack.

- Sorting
  "Kill the strong monsters first" is an example of a command that sorts available objects and then performs an action on each one.

We stress that our system is a preliminary effort. It does not implement all of the areas given above, and within the areas that are implemented our abilities are limited. Even so, we will show that our system implements enough to be a viable interface for Nethack.

In the rest of the paper, we describe the scope and implementation of our English language interface to Nethack. Section 2 describes the user interface in detail and presents examples. Section 3 gives an overview of the system as composed of a natural language interface and game engine, and Section 4 describes the three stages of natural language representation—syntax, semantics, and executable code. Section 5 traces specific example sentences through the system. Section 6 highlights unsolved problems. Section 7 gives further detail on some technical implementation issues. Section 8 offers conclusions and presents directions for future work.

## 2 User Interface

The system accepts text as input into a Prolog environment. Of course, this text can be entered by any method available to the user. A natural approach might be to use voice dictation software as a "voice to text" layer.[1]

This is, as mentioned before, a preliminary system. Of the features of a natural language input mentioned by Marilyn Walker, we can accept aspects of definite description, quantification, temporal specification, coordination, and negation.

We are quite good at definite description. A user can "kill the dog." It is also possible to use adjectives to specify the reference more precisely. For example, "drink the blue potion" is acceptable. We also accept relative clauses using the word "that." For example, we can "approach the dog that is tame."

Quantification is supported. One can specify and quantify over things, for example, "the monster" or "all the newts." In many cases, determining reference deals with metonymy: in the command "move to the potion",

[1] In our testing, the commercial package Dragon NaturallySpeaking Essentials 6.0 worked reasonably well.

"the potion" is actually referring to the potion's location; the distinction is handled transparently by the interface.

Some degree of temporal specification is available to the user through constructs for repeating an action in time. For example, "Walk left five times" and "Rest until I am healed" are possible commands. The user can also specify unbounded repetition, such as "Keep moving left", and interrupt with "Stop".

Events can be coordinated through the connectives "and", "then", and "and then". For example, the sequence "move left, drink the potion of healing, and then kill the newt" is an acceptable input.

We handle negation only for the verb "to be." For example, one can "kill all monsters that are not tame."

We cannot do any discourse reference, as our system has no memory of previous commands. We also have no rules in place for comparison. Although it would be possible to have those in the syntax, they would require a rather large knowledge base to effectively execute the comparisons.

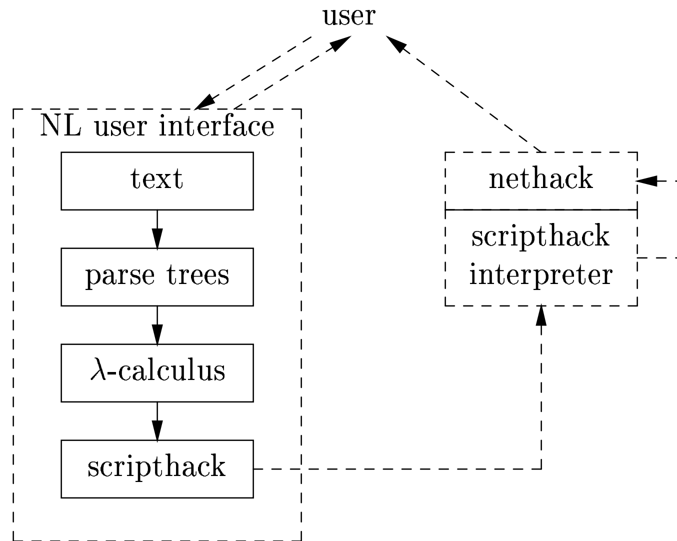
If our semantics generates more than one possible semantic representation for an input, it will prompt the user with the alternatives to resolve the ambiguity. Once one of the choices is selected, it will continue to be evaluated into Nethack commands.

We have primitive error handling that can occasionally catch grammatical errors. For example, if you input "kill all the newt" our system will inform you that you have an error of "number agreement".

# 3 System Overview

As diagrammed in Figure 2, our system is broken into two main components, running as separate processes: the natural language user interface and the Nethack game itself, augmented with an interpreter for "Scripthack," which is based on the embedded language Lua (Ierusalimschy et al., 1996). The user interface transforms spoken or written English sentences into Scripthack code, which it passes to the Nethack game process for evaluation by the Scripthack interpreter. Scripthack code can both query the state of the game and send commands to Nethack, thus carrying out the user's instructions.

Transformation from English into Scripthack occurs in three stages. A direct clause grammar (DCG) for "Nethack English" takes text and converts it into parse trees for the sentence. If a sentence has syntactic ambiguities, multiple trees are produced. Next, a denotational semantics converts each of these parse trees into logical forms based on lambda calculus. Again, multiple expressions are produced for ambigious cases. Finally, lambda calculus




Processes are represented by dotted boxes, and interprocess communication by dotted arrows. Solid boxes stand for the different representations of the user's input, and solid arrows translation between them.

Figure 2: System Organization

is converted into executable Scripthack through a deterministic (and fairly straightforward) translation.

The Scripthack code is written in a language consisting of Lua augmented with functions for accessing and modifying the state of a Nethack game, algorithms related to game-play (such as pathfinding), and facilities for higher-order functional programming. The code is executed by a Lua interpreter embedded within the Nethack process, allowing for a simple, hybrid approach to interaction with the game. Information about the game is retrieved by directly accessing memory, but the state is modified indirectly by sending normal keystroke commands as input to Nethack. This avoids some of the drawbacks of approaches that either eliminate or entrench the barriers between a natural language interface and its application: directly manipulating the game state by setting global variables is tricky and potentially dangerous, while reading the state by parsing the screen's graphics is cumbersome.

$$
\begin{array}{rcl}
s & \rightarrow & imp \mid adv \\
\\
imp & \rightarrow & vp_{in} \\
imp & \rightarrow & \textbf{keep}\ vp_{ge} \\
imp & \rightarrow & imp\ (\textbf{until} \mid \textbf{while})\ decl \\
imp & \rightarrow & imp\ conn\ imp \\
\\
decl & \rightarrow & np_{Num}\ vp_{Num} \\
decl & \rightarrow & adj \\
\\
np_{Num} & \rightarrow & pn_{Num} \\
np_{Num} & \rightarrow & (ppn \mid det_{Num})\ n_{-,Num}\ (\textbf{that}\ vp_{Num})? \\
np_{Num} & \rightarrow & (\textbf{all of} \mid \textbf{all})\ np_{Num} \\
\\
vp_{Num} & \rightarrow & v_{Inf,Num}\ c_{Inf}\ adv? \\
vp_{Num} & \rightarrow & v_{Inf,Num}\ adv?\ c_{Inf} \\
\\
pp_{Kind} & \rightarrow & p_{Kind}\ np_{-} \\
p_{to} & \rightarrow & \textbf{to} \\
\\
c_{Inf} & \rightarrow & \text{complement options for Inf} \\
\\
v_{Inf,Num} & \rightarrow & \text{verb with infinitive form Inf and number Num} \\
v_{notbe,Num} & \rightarrow & v_{be,Num}\ \textbf{not} \\
v_{Inf,Num} & \rightarrow & v_{be,Num}\ v_{Inf,ge} \\
\\
n_{Sing,Num} & \rightarrow & \text{noun with singular form Sing and number Num} \\
n_{Sing,Num} & \rightarrow & adj\ n_{Sing,Num} \\
\\
det_{Num} & \rightarrow & \text{determiner with number Num} \\
\\
ppn & \rightarrow & \textbf{my} \mid \textbf{your} \\
\\
adj & \rightarrow & \textbf{cursed} \mid \textbf{blue} \mid \ldots \\
\\
adv & \rightarrow & \textbf{yes} \mid \textbf{no} \mid \textbf{left} \mid \ldots \\
adv & \rightarrow & adv\ count \\
adv & \rightarrow & adv\ conn\ adv \\
\\
count & \rightarrow & \textbf{once} \mid \textbf{one time} \mid \ldots \\
\\
conn & \rightarrow & \epsilon \mid \textbf{then} \mid \textbf{and then} \mid \textbf{and}
\end{array}
$$

Figure 3: A simplified DCG for Nethack English

# 4 Natural Language Representation

## 4.1 Syntax

Figure 3 presents a slightly simplified version of the DCG for Nethack English, complete save for items of the lexicon and verb complement options. Below we explain some of the rules in greater detail.

- $s \rightarrow imp \mid adv$
  The sentences the interface accepts must be either imperative commands or adverbs. The latter are allowed for the case of user response to system queries. For example, "Yes" in response to "Restore saved game?"

- $imp \rightarrow$ **keep** $vp_{ge}$
  This rule allows for imperatives such as "Keep moving left" or "Keep killing the monsters."

- $imp \rightarrow imp$ (**until** | **while**) $decl$
  The interface accepts commands such as "Rest until I am healed."

- $imp \rightarrow imp\ conn\ imp$
  Imperatives can be chained together in sequence using connectives such as "and" or "then." The epsilon transition $conn \rightarrow \epsilon$ allows us to accept sentences such as "Move left, move down, and then drink the potion." However, it also causes the system to accept perhaps ungrammatical phrases like "Move left, move left."

- $decl \rightarrow adj$
  This rule assumes an implicit "I am," as in "Rest until healed."

- $np_{Num} \rightarrow (ppn \mid det_{Num})\ n_{-,Num}$ (**that** $vp_{Num}$)?
  The optional (**that** $vp_{Num}$) allows the user to enter relative clauses, for example, "Kill all the monsters that are not tame."

- $c_{Inf} \rightarrow$ complement options for Inf
  The grammar contains rules specifying allowable complements for different infinitive verbs. For example, there is a rule $c_{be} \rightarrow adj$, allowing for verb phrases like "is tame", and a rule $c_{move} \rightarrow pp_{to}$ to allow "Move to the monster."

- $v_{Inf,Num} \rightarrow$ verb with infinitive form Inf and number Num
  Save for irregular verbs, the DCG rules describing the verb lexicon are all generated automatically from infinitive forms.

| | | |
|---|---|---|
| $error(np_{Num}$, "determiner on proper noun") | $\rightarrow$ | $det_{Num}\ pn_{Num}$ |
| $error(np_{_}$, "number agreement") | $\rightarrow$ | $det_{_}\ n_{_,_}$ (**that** $vp_{_}$)? |
| $error(imp$, "verb tense") | $\rightarrow$ | **keep** $vp_{_}$ |

Figure 4: Examples of error rules

- $v_{notbe,Num} \rightarrow v_{be,Num}$ **not**
  This rule provides for a limited form of negation by treating "not to be" as a special verb, accepting sentences such as "Rest while I am not healed."

- $adv \rightarrow adv\ count$
  In our interface this rule is particularly important for movement, allowing commands such as "Move left three times then up." It sometimes produces ambiguous parsings, though, for example for the adverb "up then right twice," which can be parsed as either

  $$adv(adv(\textbf{up}), conn(\textbf{then}), adv(adv(\textbf{right}, count(\textbf{twice}))))$$

  or

  $$adv(adv(adv(\textbf{up}), conn(\textbf{then}), adv(\textbf{right})), count(\textbf{twice}))$$

- $adv \rightarrow adv\ conn\ adv$
  As with imperatives, adverbs can be sequenced, as in "Move left then up then right."

In addition to these rules, the DCG contains several *error rules* that attempt to classify syntactic errors. Figure 4 lists examples of these. Error rules work exactly as do normal DCG rules, but in addition inform the system of the existence and nature of a grammatical mistake. This allows for graceful detection, since errors do not cascade upwards. For example, the parser can still recognize "Kill all newt" as a *vp*, but notes that the *np* "all newt" has a number agreement error.

## 4.2 Semantics

The semantics of Nethack English parse trees are represented by expressions of lambda calculus augmented with constants representing lexical primitives and a variety of meta-semantic operators.

### 4.2.1 Sequencing

To coordinate two events in time, we use a two-place relation, *sequence*, which indicates that the first argument is to be evaluated before the second. For example, "rest then pray" is represented as $sequence(rest(), pray())$.

As an alternative to using *sequence*, another possible approach is analogous to the "monadic style" of functional programming. In this view, imperatives are functions from states of the world to states of the world, and likewise, for example, quantifiers are functions from states of the world to predicates to actions to states of the world. So, instead of giving two actions to the *sequence* relation, the result of evaluating the first action is passed as an argument to the second. For instance, the above example becomes $pray(rest(\sigma_0))$, where $\sigma_0$ is the initial state of the world. While this approach may be aesthetically more satisfying, it is problematic in our situation because Nethack and the Lua interpreter are both written in C, which presents a barrier to keeping track of the entire state of the world.

### 4.2.2 Adjectives

We treat adjectives semantically as predicates that can be joined to nouns to form more specific predicates. For example, "tame kitten" is a predicate satisfying objects that are both tame and kittens.

It should be noted that this representation is not always accurate. For example, the sentence "He is a good linguist but a bad man" would not be properly represented in our system, since he would have to be both good and bad. This may not be an issue for our domain, however.

### 4.2.3 Synonyms

We ease the burden of implementation by treating synonyms as having equivalent semantics. For instance, "move," "go," and "walk" all translate into the same logical form and so eventually have the same effect in the game. Likewise, both "attack the monster" and "kill the monster" are evaluated as $the(monster, \lambda m.kill(m))$.

Where words have similar meanings but some different connotation, it seems easiest and most tractable to give separate semantics. For example, "move to" and "approach" are translated into different logical forms which are eventually implemented as, in the former case, movement to a location, and in the latter case, movement right next to a location.

### 4.2.4 Subordinate Clauses

Though our grammar only allows imperatives or adverbs as sentences, subordinate clauses require dealing with declarative phrases. For example, the parse tree for "rest until I am healed" contains as a subtree the declarative "I am healed," which has a truth-functional semantics.

### 4.2.5 Scope Ambiguity

We used the Hobbs-Shieber quantifier scoping algorithm to generate all possible interpretations of statements involving quantifiers (Hobbs and Shieber, 1987). In addition to the usual issues, the Nethack semantics are complicated by the presence of imperatives, which have side-effects. For example, the sentence "rest until healed and then kill all the monsters" is ambiguous: do "all the monsters" refer to the monsters in the room before or after the adventurer finishes resting?

The approach we took to dealing with this problem was to specify that the aforementioned *sequence* relation is opaque in its second argument. The Hobbs-Shieber algorithm then generates both possible readings. For example, the above sentence is first translated into the following semantics:

$$sequence(dountil(rest, healed), kill(term(the, monster)))$$

which can be scoped in two possible ways:

$$sequence(dountil(rest, healed), the(monster, \lambda m.kill(m)))$$
$$the(monster, \lambda m.sequence(dountil(rest, healed), kill(m)))$$

## 4.3 Scripthack

Translation of logical forms into executable Scripthack is straightforward, since the language has lambda expressions as well as routines corresponding to most of the operators used in our semantics. Most of the implementation details of these routines for the Scripthack interpreter is beyond the scope of this paper, but the treatment of quantification and reference is important for understanding the NATHACK interface.

Quantification using "the" and "and" are both translated into Scripthack that eventually is interpreted in the same way, returning a list of all objects from a certain universe that satisfy a given predicate. For example, "the newts" and "all newts" are both interpreted as filters on the `isnewt` predicate over all visible monsters. Real-world English has a more subtle distinction between the universes "all" and "the" quantify over, but our treatment works

reasonably well in this domain, as the user most likely intends to refer to objects in the immediate vicinity.

A problem with this general approach is dealing with singular nouns, which naturally should only refer to a single object. If the user commands "Kill the monster" and there is more than one monster, then the sentence is most likely ambiguous and the user should be queried. Unfortunately, our system has no mechanisms in place for feedback from the game engine to the user interface, and so we use a hack. Since it seems more unintuitive for the above command to kill *all* of the monsters, in such situations we choose an arbitrary one by using a function `sg` to "singularize" predicates, that is, make them satisfiable only once (using side-effects). For example, the sentence "Kill the monsters" is translated into the following Scripthack:

```
v1=the(ismonster)
v2=(function (n1) return (kill(n1)) end)
app(v2,v1)
```

(`app` is a higher-order function that applies a function to a list of objects.) But "Kill the monster" is translated into:

```
v1=the(sg(ismonster))
v2=(function (n1) return (kill(n1)) end)
app(v2,v1)
```

# 5 Examples

We now show some sample sentences and their syntax, semantics, and Scripthack code.

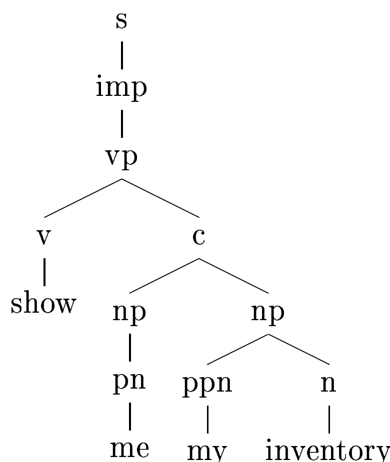
## 5.1 "Chase the Idefix"

This is an example of an ungrammatical sentence. In Nethack, "Idefix" is a proper noun; it is the name of your faithful dog companion. If an input sentence is not accepted by our DCG, we provide feedback to the user through error rules.

```
Parse error(s): determiner-on-proper-noun
```

Our DCG only allows determiners before normal nouns, so the use of a determiner followed by a proper noun results in an error.

## 5.2 “Show me my inventory”



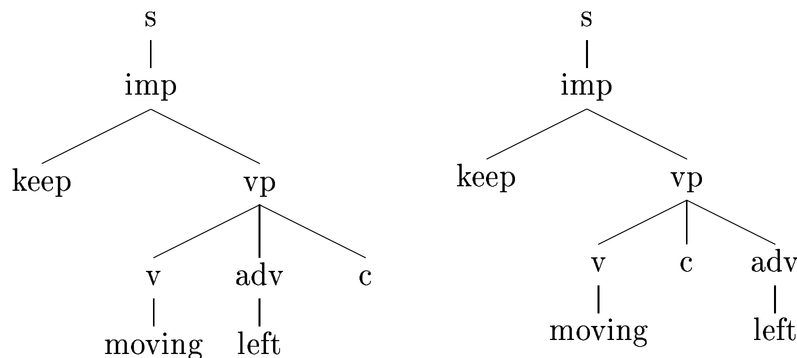

There is only one possible meaning for that phrase:

$$my(inventory, show(me))$$

Since there is no ambiguity, this form is sent immediately to the system be converted into Scripthack. It is first translated into a Scripthack tree, which is more amenable to translation from lambda calculus, and then reduced to the string of Scripthack code, `showinventory()`.

## 5.3 “Keep moving left”

This sentence has an odd syntactic ambiguity, since “moving” takes an empty complement and so the two rules $vp \rightarrow v\ c\ adv$ and $vp \rightarrow v\ adv\ c$ both apply:




However, both parse trees have the same semantics, and so the user is never prompted for disambiguation. The system continues with the following logi-

cal form:

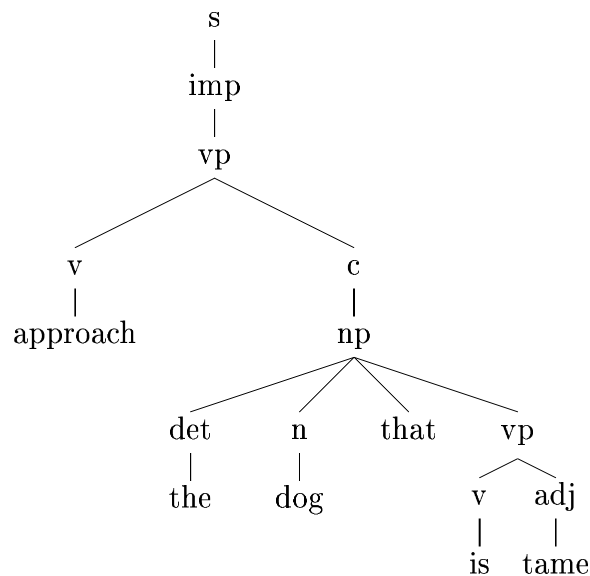
$$keep(ly(move, left))$$

Which results in the following Scripthack:

```
keep((function (x) return (move(dir_w(myloc()))) end))
```

## 5.4 “Approach the dog that is tame”

This simple relative clause has the following parse tree:




The semantics treats the “is tame” clause as a predicate modifying dog:

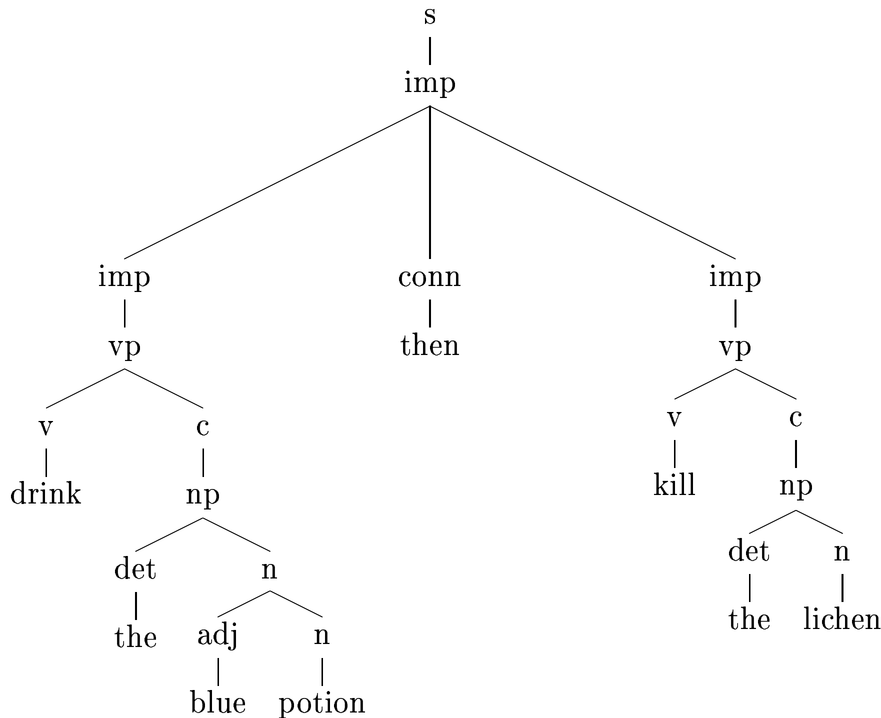
$$the(conjunct(be(tame), dog), \lambda d.approach(d))$$

which is translated into:

```
v1=the(and_pred(istame,sg(isdog)))
v2=(function (n1) return (approach(n1)) end)
app(v2,v1)
```

## 5.5 “Drink the blue potion then kill the lichen”

The parse tree for the sentence is:




This sentence is ambiguous, as was discussed in Section 4.2.5, since the reference of “the lichen” is not fixed. It could refer either to the lichen you see now while issuing the command, or else the lichen in existence after you finish drinking the potion. These two interpretations have the following semantics:

$$the(lichen, \lambda l.the(conjunct(blue, potion), \lambda p.sequence(kill(l), drink(p))))$$
$$the(lichen, \lambda l.sequence(kill(l), the(conjunct(blue, potion), \lambda p.drink(p))))$$

If the user decides to use the first meaning, it will result in the following Scripthack code.

```
v1=the(sg(islichen))
v2=(function (n2)
      v3=the(and_pred(isblue,sg(ispotion)))
      v4=(function (n1)
            local n2=%n2
            drink(n1)
            return (kill(n2))
          end)
      return (app(v4,v3))
    end)
app(v2,v1)
```

(The `local n2=%n2` line is only necessary because Lua 4.1 does not have full lexical scoping; Lua 5.0 does.)

# 6 Open Questions

We have implemented a successful implementation of a natural-language interface to Nethack that can parse and execute numerous commands. However, further improvements would be useful for a more robust and intuitive natural language interface.

## 6.1 More sophisticated error handling

As explained earlier, error handling and correction is accomplished within the syntax layer, where errant grammar is tagged within the parse tree so that an appropriate error message can be sent to the user. Unfortunately, this requires a separate DCG rule for every kind of grammatical error that should be accounted for. For this reason, our own implementation of error-correction deals with a rather small subset of possible user mistakes.

This method is labor intensive, and unless Nethack operates on a sufficiently small subset of grammar, it may be preferable to find some other method of error correction.

Moreover, we have not addressed the interface question of guiding a user to correct input. When confronted with a natural language system, users tend to type all sorts of things outside the expectation or experience of the designers. An interface should give feedback on its limitations and on proper format for input, so a user can learn and adapt. In our case, we give only limited information on a small class of parse errors to the user; improving and broadening our error handling is an important open problem.

## 6.2 Semantic errors

The command "kill the coins" is grammatically correct, but semantically incorrect. Finding semantic errors is an important open question.

In a way similar to our handling of grammatical verb/object agreement via subcategorization, it would be possible to subcategorize verbs and their objects for semantic agreement. Thus, the verb "kill" could be marked such that only accepts "animate" complements, and "coin" would not fulfill that requirement.

## 6.3 Queries

Most actions are performed by the user to objects in the world, but queries are also a critical part of a player's interaction with the environment. We currently deal with queries phrased as imperative commands, such as "Show me my inventory", but valid queries may also take the form of interrogative sentences, such as "How much money do I have?". Handling this would require the creation of a separate syntactical form. This would require extensive work but should nonetheless be tractable.

## 6.4 Pronouns

One of the key characteristics of natural language is use of pronouns to refer to previously mentioned objects. (For example, "move to the monster and kill it".) One simple (and incorrect) method of pronoun interpretation is using the most recent noun in place of the pronoun. A more sophisticated notion of discourse theory is required to find correct interpretations of pronoun phrases.

## 6.5 Context

Many phrases are ambiguous to our system. For example the command "kill the monster" causes our system to arbitrarily pick the closest monster and attempt to attack it. We propose two possible means of disambiguating unclear sentences. First, a system should refer to the context of the previous discourse with the user. Second, the system should have access to the context of the state of the world.

For example if a user had issued the command "attack the blue jelly" two turns ago, it is fairly likely that a command like "kill it" or "kill the monster" would continue to refer to the blue jelly. To refer to that user's discourse, the system would require a working memory that could be searched for previous informative user input. The earlier discussion of verb subcategorization could also prove helpful here, because one would know what nouns would be acceptable replacements for the ambiguous entries.

The ambiguous command "kill the monster" can also be resolved by knowledge of the state of the world. If there is only one monster, there is no actual ambiguity. Even though it is possible for "the monster" to mean many different creatures, there is only one that it is currently possible for the user to kill.

We can also imagine the existence of a knowledge base that would allow reasoning about what a sentence means intuitively, or what choices might be clearly better for game play. A sentence such as "Kill the dragon with

the sword" is ambiguous, because it is not known whether the sword is being wielded by the dragon or whether the sword ought to be used to kill the dragon. A knowledge base could inform us that dragon do not carry swords, thus solving our ambiguity. A pure first order logic combined with a theorem prover, such as Otter, might be useful for analyzing this knowledge base. Here, however natural-language methods intersect the world of artificial-intelligence, and we find ourselves beyond our realm of knowledge.

# 7 System Details

## 7.1 Parsing

Our DCG interpreter supports limited regular expressions and is based on the CKY bottom-up parsing algorithm. It performs memoization to cache all the parse trees generated for a string.

## 7.2 Nethack and the Scripthack Interpreter

Nethack and the Scripthack interpreter reside in the same process as separate threads, communicating through Unix pipes and using semaphores for synchronization. The specifics of the system are beyond the scope of this paper.

## 7.3 Implementation Status and Availability

We have implemented the system described above. You can find our work at: `http://hcs.harvard.edu/~stsfp/nathack.tar.gz`

# 8 Conclusion

Our work demonstrates the viability of natural language interfaces for rogue-like adventure games. It would be interesting to extend this approach to other sorts of games. In particular, can natural language be used for coordination between human and computer players in a 3-D environment, such as id software's Quake? We invite further work on pushing natural language beyond Infocom style games.

# 9 Acknowledgments

Professor Shieber and Ken Shan provided advice throughout the building of this project. Professor Ramsey made many helpful comments on software engineering and pointed us to ROG-O-MATIC. We are grateful to the maintainers of Nethack for giving us a fine system to play with. Finally, we acknowledge Idefix, our little dog, who dies very often.

# References

Crowther, W., D. Woods, and D. Knuth (2001). Adventure source code in CWEB. `http://www.literateprogramming.com/adventure.pdf`.

Hobbs, J. and S. Shieber (1987, January - June). An algorithm for generating quantifier scopings. *Computational Linguistics 13*(1-2), 47–63. `http://www.eecs.harvard.edu/~shieber/papers/quantex.pdf`.

Ierusalimschy, R., L. H. de Figueiredo, and W. Celes (1996, June). Lua — an extensible extension language. *Software — Practice and Experience 26*(6), 635–652. `http://www.lua.org/spe.html`.

Koller, A., R. Debusmann, M. Gabsdil, and K. Striegnitz (2002). Put my galakmid coin into the dispenser and kick it: Computational linguistics and theorem proving in a computer game. Journal Submission.

Mauldin, M., G. Jacobson, A. Appel, and L. Hamey (1984). ROG-O-MATIC: A belligerent expert system. In *Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*. `http://www.cs.princeton.edu/~appel/papers/rogomatic.html`.

Waijers, B. (2003). The roguelike games home page. `http://www.win.tue.nl/~kroisos/roguelike.html`.

Walker, M. (1989). Natural language in a desk-top environment. In *Proceedings of HCI89, 3rd International Conference on Human-Computer Interaction*, pp. 502–509. `http://www.research.att.com/~walker/ab9.ps`.