Summary of the ALFRED 2.0 Project

Scott Fults
scott.fults@gmail.com
June 2, 2011

1. Introduction

ALFRED 1.0 was a universal interfacing agent that accepted a very small set of English sentences and translated them into commands appropriate for different domains. ALFRED itself had no parsing capacity (parsing was conducted by an outside, off-the-shelf system), and its knowledge of language in general was extremely limited. It basically searched for keywords in an input string and matched them up to possible domain commands.

ALFRED 2.0 is the next generation of this interfacing agent. ALFRED 2.0, like its predecessor, is written in Alma. Alma is a Prolog-like, declarative language (Alma itself is written in Prolog). What makes Alma unique is its step-by-step reasoning. Alma keeps track of time by counting off steps (or time units). Each step in a derivation must correspond to an Alma step. For instance, at step 1, Alma can accept that P, P -> Q, and Q -> R is true. Then each step in the derivation of R takes a time step:

1) step 1: P
P -> Q
Q -> R
step 2: Q
step 3: R

In a language like Prolog, R would be concluded right away. This step-by-step reasoning was designed to accommodate contradictions in an agent's knowledge base. (Purang 2001, Josyula 2005)

Thus, ALFRED 2.0 is a semantic ontology, i.e. a characterization of knowledge, written in the Alma formalism. Alma is the reasoning engine that, step-by-step, expands/changes ALFRED's knowledge base (KB). This KB consists of linguistic knowledge (mostly) but was designed so that general knowledge could be added easily. ALFRED 2.0's linguistic ontology is discussed in sections 2 and 3.

ALFRED 2.0 was designed to contain all of the functionality of the first ALFRED, but with enhanced natural language and conceptual reasoning skills. As such, ALFRED 2.0 was created to provide a natural language, bidirectional interface to an open set of domains. It acts as a translator, taking typed English utterances as input (although it can also take voice input from a speech-to-text software package called Dragon), and outputs the domain appropriate command string. The idea is that ALFRED 2.0 is general enough that it can function this way with any domain, provided it is supplied with the appropriate grammatical information about the domain language and the English sentences that would be used as input. Example grammars of this sort are discussed in section 4.

ALFRED 2.0 was also designed to be a general, common sense reasoner. The logical forms generated by an incoming utterance or used to generate an outgoing utterance are the

same logical sentences that ALFRED can use to reason about the world.

In addition, ALFRED 2.0 was to be structured such that a separate software application called MCL (MetaCognitive Loop) could be easily added on top of it. MCL is a self-monitoring software package that provides a system with the metacognitive ability to notice anomalies, analyze them, and respond to them. In ALFRED's case, these would most likely be communication and/or language related anomalies. This type of metacognition can result in modification of conceptual knowledge and linguistic knowledge.

Metacognitive capabilities require that ALFRED 2.0 be designed such that its linguistic knowledge is represented in the same formalism as all other knowledge. That is, there can be nothing "special" about its linguistic knowledge: it must be accessible in every way that other knowledge is accessible. In ALFRED 2.0, for instance, linguistic objects are given the status of first-class objects that can have properties and stand in relations to other objects. This crucially means that logical sentences can be created with linguistic objects (such as words, letters, categories, rules) that ALFRED can then reason about.

One further motivation for this ontology design is that it allows dialog about linguistic items. Metacognition and dialog management are closely related. Dialogs often contain discussion of the dialog (e.g., clarification questions), and if ALFRED 2.0 is to engage in metacognition it may be necessary to also engage in metadialog with the user.

2. Design Specifications of ALFRED 2.0

2.1 Overview of design

ALFRED 2.0 was designed to function like a machine translator. In fact, we have used a classic machine translation design with an interlingua. ALFRED takes an input string, attempts to translate the string into a kind of first-order logic (the interlingua), and then translates that into the appropriate output language. For instance, if the input is the English sentence "Recharge at waystation0", then this would be translated into "charging(e) & location(e,x) & waystation0(x)" where existential quantification over the variables is assumed. This is then translated into "charge() loc(0)" for our Mars Rover domain. ALFRED 2.0 is designed so that translation going in the other direction, from domain to user, is also possible. This is the standard "pipeline" design of linguistic processing: a series of processes are applied, with the output of one process serving as the input of the next.

ALFRED's linguistic knowledge can be thought of as consisting of two main parts: static linguistic knowledge containing knowledge of the languages it knows such as English and the domain languages; and utterance knowledge consisting of knowledge about a dialog it is currently having or has had in the past. We will discuss static knowledge below in section 3. This section concerns the processes that ALFRED 2.0 uses to understand a current utterance/ dialog.

ALFRED 2.0 has several processes involved in going from English to domainese (and vice versa). First, the input string is parsed. Then it is turned into a logical form. Then a new string is generated. We discuss these processes below. These processes are controlled by a dialog management system.

2.2 Dialog management

The current system in ALFRED used for dialog management is very basic. But the

system was designed such that a more sophisticated system could be put in place with minimal changes to the other subsystems in ALFRED. Right now, ALFRED 2.0 keeps a queue of incoming utterances and processes them one by one. The utterances at the top of the queue becomes the current utterance. Only current utterances are processed. Once a current utterance has been processed, it is marked as Done and the next utterance in the queue becomes the current utterance.

A dialog is a first-class object in that there are things that are dialogs. Thus a dialog can have participants (who can then have turns, etc.). Dialogs also have utterances.

2.3 Tokenization

Once an utterance becomes a current utterance, it is tokenized. This process is done in Prolog (i.e., the Carne part of the Alma/Carne system). This means that it is not done in a stepwise fashion and its parts are not analyzable within the Alma system. ALFRED 2.0 sends the utterance string off to Carne, and in the next few steps (from ALFRED's point of view), the utterance gains several properties. Carne splits the utterance up into words, which the utterance "has". And, each word is split up into letters, which again each word "has". For example, utterance s1 might have the following relationship added to the KB:

2) isa(w1, word)
has(s1, word, w1)

(2) should be read as "w1 is a word" and" s1 has word w1". Furthermore, w1 will have spellings, which are also given first-class status. Spellings in turn have letters, and letters have ascii-codes.

3) isa(sp1, spelling)
has(w1, spelling, sp1)
isa(ltr1, letter)
has(sp1, letter)
has(ltr1, ascii_code, 112)

This information is all added to the KB by Carne. But this is enough information to start the parsing process.

2.4 Lexical Look-up

Lexical look-up is facilitated by the ALFRED's static knowledge of a languages dictionary and its alphabet. Again, this linguistic knowledge is given the status of first class objects, hence "there is an alphabet" and this alphabet "has letters" and each letter "has an ascii code" associated with it. Each lexical item has a spelling, and each spelling has letters that are linked up to the alphabet. Therefore, the lexical look-up process is a search that starts with finding the lexical item that is linked to the same letters in the alphabet in the same order as the utterance word. Once this is found, the utterance word is linked to the lexical entry and any information that carries over to the utterance word is copied. For instance, the syntactic category of the lexical entry is copied over to the utterance word (e.g., if the lexical entry is a verb, then the

utterance word is a verb.)
This is repeated for each word in the utterance.

2.5 Parsing

After each utterance word is assigned a part of speech, the parsing starts. It was unclear to us if Alma was capable of doing backwards searching with all the functionality of Prolog. Hence it seemed probable that a simple Definite Clause Grammar was impossible to do. So, we opted for a chart parser based on an Earley algorithm. It is a top-down, breadth-first parser. The parser consists of a set of context free grammar rules (part of the static linguistic knowledge), a predictor, scanner, completer and a structure builder. We talk about each in turn.

The parser creates states based on the context free grammar rules and any previous states. The context free grammar rules have a left hand side (LHS) and a right hand side (RHS) and are basically simple rewrite rules. For example, S → NP VP should be read as "S is rewritten as NP VP". The rules are first-class objects in ALFRED's KB. The parser reads the RHS from left to right, and when it comes across a new symbol, it tries to find that symbol in another rule (on the LHS) or in the part of speech of some utterance word. If it finds that symbol, it "consumes" it, and then looks for the next symbol. This is typically represented with a "dotted rule":

(4) S → .NP VP
S → NP.VP
S → NP VP.

The dot represents where in the rule the parser is currently at. In the last example of (4), the parser has consumed everything on the RHS. We represent this information with two lists, one for the left of the dot and one for the right; or, one list for the consumed symbols and one for the unconsumed symbols:

(5)	S → .NP VP	[ ]	[NP, VP]
	S → NP.VP	[NP]	[VP]
	S → NP VP.	[NP, VP]	[ ]

This information is kept in a state. Each state has the following information:

(6) Parsing state:
(state #, LHS of rule, consumed RHS, unconsumed RHS, consumed words, unconsumed
words, history)

The start sequence generates the seed state. The predictor uses the rules to generate new states from old states. First, it looks at the list of unconsumed RHS in each previous state. If the next item can be expanded by a context free grammar rule because that item is on the LHS of the rule, then it can create a brand new state out of that rule. The new state that has:

(6) a new state id
a LHS (according to the new rule)
an empty consumed RHS (because the rule is just starting to be used)
an unconsumed RHS with all of the items in the rule's RHS
uses the old rule's processed and unprocessed word lists
an empty history list, since the history encodes daughters and there aren't any yet

The scanner looks at the unprocessed RHS of states and if the next item is a part-of-speech, it checks the current word to see if its part-of-speech matches. If it does, then it creates a new state with:

(7) a new state id
the part-of-speech tag as the LHS
the word id as the consumed RHS
an empty unconsumed RHS
the word as its processed word list
remaining unprocessed words from old state as the new unprocessed word list
history is empty

The completer looks at old states that have no items in the unconsumed RHS list, i.e. a finished state. It then tries to match that state's LHS to a previous unfinished state that has the same symbol at the top of its RHS. For example, if a state has completed an NP (NP → N PP.), and there is a previous state that needs an NP (VP → V . NP), then it will create an updated version of the previous state by moving the dot over and consuming the NP (VP → V NP.) . Furthermore, the processed word list of the finished state must be on top of the unprocessed word list of the unfinished state.

In other words, when one state signifies that a rule has been completed, all states needing that rule have their RHS progressed by one item. This is done by moving the item from the unprocessed RHS list to the processed RHS list. This, in turn, can trigger the Predictor or the Scanner to apply to the new state.

The Completer also keeps a history of the derivation by adding the state numbers of the finished state (the daughter of the unfinished state) to the history list of the new state. The Completer creates a new state that:

(8) a new state ID
has the same LHS as the unfinished state
has unfinished state's processed RHS list appended with the top member
of its unprocessed RHS list
has the top member of unfinished state's unprocessed RHS list removed
has unfinished state's processed word list, appended with the finished state's processed word list
has the unfinished state's unprocessed word list but with the finished state's processed word list removed from the top
adds the state number of the finished state to the history list

The parse is completed (and legal) when it completes the state Start → S and all of the words have been processed.

After a legal parse is found, the Structure Builder follows the derivation by reading the history lists, starting with the top S-node, down to all of the terminals, building the structure as it goes. The result is a styntax tree with phrases, constituents, terminal, etc., which each have first-class status as objects.

2.6 Theta-role assignment

Theta-roles are syntactic objects that are assigned by terminals according to their lexical requirements to sister constituents in a syntactic tree. They have semantic effects, but they are purely syntactic. The theta-role assigner simply assigns a theta-role form a terminal element to its syntactic sister, if and only if, that terminal element is specified in the lexicon as a theta-role assigner.

2.7 Building the logical form

The logical form is a logic sentence built with the primitives used in ALFRED's general common sense ontology. ALFRED has concepts of places like waystations and runways, agents like rovers and planes, and events like chargings of batteries, circlings of planes and movings. When it makes a logical form out of an utterance, it takes tokens of these concepts and fits them into a logical sentence.

The logical form is built from the bottom of the syntax tree up. It starts with what we call the linking rules. These linking rules find all of the terminals in the tree and create logical form counterparts by finding the concepts that coincide with the terminal elements (i.e. the lexical entries from the utterance). This simply means that a logical phrase is created, and it is linked to a token of the meaning it is associated with. Also, the syntactic phrase is semantically linked to the logical phrase.

(9) isa(p1, logical_phrase)
has(concept1, token, p1)
has(synp1, semantic_value, p1)

Then, ALFRED goes up the tree using one of four combinatorial rules to build new parts to the logical form. The rules correspond to possible subtrees in the syntactic tree built previously.

(10) Semantic Combo Rule 1: If the subtree is like A → B, where A does not have a theta-role,
then pass up the meaning of B without changes.

This just means that there is no new logical phrase created. The top syntax phrase is also related to the same logical phrase as the syntactic daughter.

(11) Semantic Combo Rule 2: If the subtree is like A → B C where B is vacuous and C is a constant or a predicate, then pass up the meaning of C without changes.

Again, this just means that there is no new logical phrase created, and instead A is linked to the same logical phrase as C.

(12) Semantic Combo Rule 3: if the subtree is like A → B C and B and C are both predicates, then make a new logical phrase that is B and C conjoined.

This simply means that a new logical phrase is created with daughters that are the semantic values of B and C. The conjunction is assumed.

(13) Semantic Combo Rule 4: if the subtree is like A → B and A has a theta-role, then create a new logical phrase that is a 2 place predicate which is a token of the theta-role meaning, an event variable and a variable linked to the constant, i.e., "THETA(e, B)"

This means that a new logical phrase is created that corresponds to "THETA(e,B)" where THETA is a token of the theta role meaning, "e" is an event variable, and "B" is the constant associated with the B subtree. It's daughter is the semantic value of B.

These four rules create a logical form tree that corresponds to the syntactic tree. The idea is that it also creates a logical form that can be reasoned about *in ALFRED's conceptual space* (but it seems that we have failed in this, see below).

We should note, also, that the current logical form built from an utterance is incomplete in that it does not include any speech act information nor does it include an agent information. For instance, if the utterance is "recharge at waystation0", the logical form built is something like "charging(e) & location(e, x) & waystation0(x)", where the existential operators are understood. This is clearly not an accurate logical form for the *command* "recharge at waystation0" which would also contain agentive information such as "agent(e, y) & rover(y)" describing the agent of the event, and speech act information such as "want(z,e) & user(z)", meaning something like the user wants *e* to happen, or some variation. Also not included is time and aspect information, etc. But we felt that this part of the logical form could be added on top of this basic logical form after this was implemented.

2.8 Generation

Generation is the creation of string in some language that follows that languages context free grammar rules and is made up of words in that languages lexicon. It starts with a logical form, i.e., a concept that ALFRED desires to express in a language. We will continue with the example of a user utterance, in English, that is translated into a logical form and then generated into a domain language string.

Of all the processes involved in the linguistic processing pipeline that makes up ALFRED 2.0, the generation of a string is the most complicated and cumbersome. One might think that generation is simply parsing done backwards, but we have had considerable trouble implementing that idea.

We use what is essentially a modified Earley algorithm again, i.e., a chart parser. It has been modified to use an unordered list of logical phrases (those that make up the logical form

discussed in the previous section) rather than an ordered list of words as the parser does. This of course makes sense: when parser, we start with an ordered list of words that we must give structure to, but when generating we start with a bunch of concepts which may appear in any order that the language requires or permits. Complications arise from the fact that the input is not an ordered word list.

Essentially the generator must build subtrees (i.e., states) that, when we apply the semantic combo rules to them, create the bits of the logical form. In other words, we must include not just the what the parser did, but also the structure building, and logical form building in the generation process. That means we must include the logical form building *in the state building process of the Earley algorithm*. Incidently, nothing is really done "backwards" here. The reasoning is forwards (i.e., the same as it was in the parsing process), but it is checked against a logical form rather than the word list. There is a major design flaw in how we went about this, which I will wait to discuss until the end of this section.

First, we find all of the logical form phrases created by the logical form builder discussed in the previous section. These make up the unprocessed list in the first state. We include much more information in each state.

(14) Generation start state:
state(utterance #,
state #,
LHS,
consumed RHS,
unconsumed RHS,
processed logical form phrases,
unprocessed lf phrases,
history,
needs_theta,
gives_theta,
semantic_check_rule,
daughter_lfs)

The utterance and state numbers are unique identifiers of the state and the utterance it is associated with. The LHS is the left-hand side of the context free grammar rule being used. The consumed and unconsumed RHS is the right-hand side of the rule with how much of it has been consumed. The logical form phrases are the lists of phrases in the logical form. The history is the state's daughters. Needs_theta is a list of any items on the RHS that have been consumed but still need to be assigned a theta-role. This occurs when an item in the logical form phrases has a theta-role meaning in it. Gives_theta is a list of all logical form phrases that have been processed that need to assign a theta, but have not yet been able to do so. The semantic check rule is the semantic combo rule that should be applied to the subtree, either 1, 2, 3, or 4. The daughter lf list is a list of the logical form phrases that correspond to the items that have been consumed on the RHS. Another parameter that is stored is the mother lf. It is kept in a separate predicate (but could have just as well been added onto the end of the state predicate. The mother lf is the logical form associated with the LHS symbol.

First, a start state is created for Start → .S

The generator's predictor is similar to the parsing predictor. If there is a state with an unconsumed RHS, and the first symbol matches the LHS of a context free grammar rule then make a new state out of that rule. The only real difference is that a semantic rule number is added to the state (semantic rule checker), which the system gets from the representation of the context free grammar rule.

The scanner has two parts. The first is the vacuous case. In this case, the syntactic node being built is vacuous semantically. Thus there is nothing in the logical form list to match it to. The scanner checks on the lexicon to see if there is an item that can fit in that slot that is semantically nil. If so, it builds a new state with this item, doing nothing to the logical form list.

The second case is when there is a lexical item that fits in the syntactic slot, and is associated with a meaning that could have a token in the logical form list. If this is the case, create a new state by moving the logical form item over to the processed list. If the item assigns theta-roles, then add it to the Gives_theta role column.

The completer is quite cumbersome. We reproduce the algorithm here:

(15) The (Corrected) Completer Algorithm

```
CS = Completed State (e.g. X -> Z.)
IS = Incomplete State (e.g. XP -> .XY)
MIS = Modified Incomplete State
1. Find CS [ X -> Z. ]
2. Find IS [ XP -> .XY ] or [XP -> .X]
3. If X = needs-theta in CS
        Calculate MIS [XP -> X.Y] or [XP -> X.]:
                needs-theta = X-LF
                daughters = [+ ] (append nothing to daghters from IS)
Else if X = gives-theta in CS
        Calculate MIS [XP -> X.Y] or [XP -> X.]:
                gives-theta = X-theta
                daughters = [+ X-LF]
Else
        Calculate MIS [XP -> X.Y] or [XP -> X.]:
                daughters = [+ X-LF]

1. Find CS [Y -> W.]
2. Find IS [XP -> X.Y]
3. If Y = needs-theta
        If X = gives-theta in IS
                MIS [XP -> XY.]
                match theta-roles
                remove the needs theta LF from unprocessed-LF, append it to daughters
        Else (X is neither needs-theta nor gives-theta)
                MIS [XP -> XY.]
```

needs-theta = Y-LF
daughters = [+]
Else if Y = gives-theta
If X = needs-theta in IS
MIS [XP -> XY.]
match theta-roles
remove the needs theta LF from unprocessed-LF, append X-LF to daughters IN PROPER ORDER (X then Y)
Else (X is neither needs-theta nor gives-theta)
MIS [XP -> XY.]
gives-theta = Y-theta
daughters = [+ Y-LF]
Else (Y is neither needs-theta nor gives-theta)
MIS [XP -> XY.]
daughters = [+ Y-LF]

When ever a state is build with nothing left to be consumed, then the semantic checker kicks in. The semantic checker takes the semantic checker rule number, and basically applies one of the 4 combo rules to the subtree. It checks to see if the result is in the unprocessed lf list. If it is, then it makes a new state. If it is not, then nothing happens.

The generator is complete when it builds a state with a completed start rule and all of the logical form phrases have been processed. It then builds a syntactic tree, in much the same way as before, using the history column. This is then linearized, with lexical items looked up in the correct dictionary.

The string is sent to the domain and we loop back to the dialog manager, which marks the utterance as done and pops off the top of the new utterance queue to be the next current utterance.

3. Implementation

This section discusses the Carne routines in ALFRED 2.0. Carne is essentially an external processor of Alma. It takes Prolog calls, processes them, and returns any results to Alma. It can also insert and delete formulas into the Alma KB. Alma differentiates between what are called Prolog calls and prolog routines. We disucss prolog routines first.

3.1 Prolog routines

Prolog routines for ALFRED 2.0 are located in the file called routines.pl. This file must be in the /src directory for ALFRED to run properly. It contains simple and relatively quick Prolog routines, written in Prolog, that ALFRED needs to complete. Some of the routines in this file were left over from the previous version of ALFRED, and we are unsure whether Alma needs them or not. The new ones include: discard_duplicates, cappend_num_to_string, cappend_string, cappend_list, chars_to_atoms.

Discard_duplicates is run from ALFRED 2.0 with an eval_bound command. It is used to avoid creating duplicate states in the parsing and generation process. This is a danger due to a quirk in Alma's step-by-step processing. Before creating new states, the parser and generator

check to see if one already exists with the same parameter values. If it does, then the new state isn't created (this is actually the advantage of a chart parser). But, if two or more identical states will be created *in the same step*, then the checking procedure doesn't catch any of them. Each would be created, and since each have different state numbers, they aren't ever considered to be duplicates. Hence, the system first creates a bunch of formulas that are not states (but only in name), gathers them all and sends them to discard_duplicates.

Cappend_num_to_string, cappend_string, cappend_list each append by using simple Prolog append tools. Chars_to_atoms is also simple Prolog tool that takes characters and turns them into ascii numbers.

3.2 Prolog calls

Prolog calls are written in their own files. They are also each stored in the /src folder. ALFRED 2.0 uses lexeme_speller, token_utt_comma, and token_utt.

Token_utt takes an utterance string, in ascii characters, and splits it up into words and letters. It asserts that each word exists, and also asserts each word's spellings and the letters contained in those spellings.

Token_utt_comma takes an utterance that contains two utterances split up by a comma such as "No, recharge at waystation0" and splits it up into two utterances.

Lexeme_speller takes words entered in the lexicon, tokenizes them into constituent letters, and then links them up to the alphabet.

4. Grammars, static linguistic knowledge

ALFRED 2.0 was designed to be general in that it shouldn't matter which language an input was in, it could be parsed and translated into logical form. Then that logical form could be translated into another language. The input could be English and the output Roverese, or vice versa. It was also designed to be general in the sense that any language could be added to ALFRED's knowledge base, provided it were supplied in the right format. This section discusses that format and how to add a language to ALFRED 2.0.

4.1 How to build a grammar for use with ALFRED 2.0

The easiest way to go about this is to define a domain grammar in the usual ways. This would generally be supplied by the domain's designer, i.e., whoever created the necessary protocols for the domain system. Then, you must make a list of all possible domain commands. This is generally a finite number.

Given this list, create another list of English phrases, sentences, or commands that might be uttered as equivalent to the domain's commands. For instance, if the domain command is "charge() loc(0)", this might correspond to "charge at waystation zero". It is best to try to limit the number of English phrasal types, and sometimes this requires some creativity to get a good English sentence. This is good practice in order to narrow down the grammar.

Finally, link each English and domain utterance together with a possible logical form. A good working knowledge of how ALFRED does its translation will help with this process. From this final list, you must create a grammar for English and a grammar for the domain language. If ALFRED already has a grammar for English, then you must add what is needed for the new domain.

Each grammar consists of four things: lexicon, context free grammar rules, theta-roles, and categories (parts-of-speech). In addition, meaningful items must have concepts that they correspond to. For instance, lexical items typically correspond to a concept, such as the word “move” belonging to the concept of moving.

Categories are like parts of speech. In English, they are things like verbs, nouns, prepositions, etc. In domain languages, they can be whatever you feel is necessary. Theta-roles correspond to semantic roles, or participants in events. Examples include agent, patient, and location.

Context free grammar rules generate the sentences in your list. They MUST consist of a mother and one or two daughters (no more). The mother is called the left_hand_side of the rule. The daughter(s) are a list on the right_hand_side fo the rule. Each rule must be assigned to a semantic checker rule number telling the parser/generator which semantic combo rule to use.

The lexical items are created using a simple predicate with seven parameters: id #, language id # that the word belongs to, its string value, the category it belongs to, the meaning id# that it corresponds to, whether it is a predicate or a constant, and any theta-roles it assigns.

These grammar simply need to be inserted into ALFRED’s code at the appropriate spots.

4.2 Sample grammars

Here is an example of an English grammar that works with the plane domain and the Mars rover domain:

```
/*English for Planes and Mars:*/
/*Build a domain.*/
isa(d0, domain, d0).

/*Build a language and make it part of the domain.*/
isa(d0, language, c2).
has(d0, language, d0, c2).
/*Make categories*/
/*
c2cat1 -- c2cat3
*/

isa(c2, category, c2cat1).
has(c2, category, c2, c2cat1).
has(c2, label, c2cat1, v).

isa(c2, category, c2cat2).
has(c2, category, c2, c2cat2).
has(c2, label, c2cat2, n).

isa(c2, category, c2cat3).
```

```
has(c2, category, c2, c2cat3).
has(c2, label, c2cat3, p).

/*Domain Syntax Rules:
*/
/*
c2syntax1 -- c2syntax6
*/

/*S --> VP*/
isa(c2, domain_syntax_rule, c2syntax1).
has(c2, domain_syntax_rule, c2, c2syntax1).
has(c2, left_hand_side, c2syntax1, s).
has(c2, right_hand_side, c2syntax1, [vp]).
has(c2, semantic_checker_rule, c2syntax1, scr1).

/*VP --> V . NP*/
isa(c2, domain_syntax_rule, c2syntax2).
has(c2, domain_syntax_rule, c2, c2syntax2).
has(c2, left_hand_side, c2syntax2, vp).
has(c2, right_hand_side, c2syntax2, [v, np]).
has(c2, semantic_checker_rule, c2syntax2, scr3).

/*VP --> V*/
isa(c2, domain_syntax_rule, c2syntax3).
has(c2, domain_syntax_rule, c2, c2syntax3).
has(c2, left_hand_side, c2syntax3, vp).
has(c2, right_hand_side, c2syntax3, [v]).
has(c2, semantic_checker_rule, c2syntax3, scr1).

/*VP --> VP PP*/
isa(c2, domain_syntax_rule, c2syntax4).
has(c2, domain_syntax_rule, c2, c2syntax4).
has(c2, left_hand_side, c2syntax4, vp).
has(c2, right_hand_side, c2syntax4, [vp, pp]).
has(c2, semantic_checker_rule, c2syntax4, scr3).

/*NP --> N*/
isa(c2, domain_syntax_rule, c2syntax5).
has(c2, domain_syntax_rule, c2, c2syntax5).
has(c2, left_hand_side, c2syntax5, np).
has(c2, right_hand_side, c2syntax5, [n]).
has(c2, semantic_checker_rule, c2syntax5, scr4).
```

```
/*PP --> P NP*/
isa(c2, domain_syntax_rule, c2syntax6).
has(c2, domain_syntax_rule, c2, c2syntax6).
has(c2, left_hand_side, c2syntax6, pp).
has(c2, right_hand_side, c2syntax6, [p, np]).
has(c2, semantic_checker_rule, c2syntax6, scr2).

/*Make concepts for English theta-roles*/
/*
c2theta1 -- c2theta3
meanstheta1 -- meanstheta3
*/

/*Patient*/
isa(c2, theta_role, c2theta1).
has(c2, theta_role, c2, c2theta1).

/*Location*/
isa(c2, theta_role, c2theta2).
has(c2, theta_role, c2, c2theta2).

/*Destination*/
isa(c2, theta_role, c2theta3).
has(c2, theta_role, c2, c2theta3).

/*Make concepts for event_participants, i.e., meanings of theta-roles*/
/*Patient*/
isa(meanstheta1, meaning, meanstheta1).
isa(meanstheta1, event_participant, meanstheta1).
has(c2theta1, content, c2theta1, meanstheta1).

/*Location*/
isa(meanstheta2, meaning, meanstheta2).
isa(meanstheta2, event_participant, meanstheta2).
has(c2theta2, content, c2theta2, meanstheta2).

/*Destination*/
isa(meanstheta3, meaning, meanstheta3).
isa(meanstheta3, event_participant, meanstheta3).
has(c2theta3, content, c2theta3, meanstheta3).

/*Lexemes:
c2lex1 -- c2lex27
means1 -- means22
```

```
*/

/*For Planesh:*/
/*---- Verbs ----*/
/*report*/
do_add_lexeme(c2lex1, c2, report, c2cat1, means1, predicate, c2theta1).

/*circle*/
do_add_lexeme(c2lex3, c2, circle, c2cat1, means3, predicate, c2theta1).

/*disconnect*/
do_add_lexeme(c2lex6, c2, disconnect, c2cat1, means6, predicate, c2theta1).

/*land*/
do_add_lexeme(c2lex7, c2, land, c2cat1, means7, predicate, c2theta1).

/*close*/
do_add_lexeme(c2lex26, c2, close, c2cat1, means22, predicate, c2theta1).

/*---- Nouns ----*/
/*Grammar*/
do_add_lexeme(c2lex2, c2, grammar, c2cat2, means2, constant, nil).

/*Plane33*/
do_add_lexeme(c2lex4, c2, plane33, c2cat2, means4, constant, nil).

/*Plane44*/
do_add_lexeme(c2lex10, c2, plane44, c2cat2, means9, constant, nil).

/*Planes*/
do_add_lexeme(c2lex5, c2, planes, c2cat2, means5, constant, nil).

/*Approach Path 22*/
do_add_lexeme(c2lex9, c2, path22, c2cat2, means8, constant, nil).

/*Approach Path 11*/
do_add_lexeme(c2lex11, c2, path11, c2cat2, means10, constant, nil).

/*Paths*/
do_add_lexeme(c2lex27, c2, paths, c2cat2, means5, constant, nil).

/*---- Prepositions ----*/
/*at*/
do_add_lexeme(c2lex8, c2, at, c2cat3, nil, nil, c2theta2).
```

```
/*For Roverese:*/

/*---- Verbs ----*/
/*charge*/
do_add_lexeme(c2lex12, c2, charge, c2cat1, means11, predicate, nil).

/*recharge*/
do_add_lexeme(c2lex13, c2, recharge, c2cat1, means11, predicate, nil).

/*acknowledge*/
do_add_lexeme(c2lex14, c2, acknowledge, c2cat1, means12, predicate, nil).

/*localize*/
do_add_lexeme(c2lex15, c2, localize, c2cat1, means13, predicate, nil).

/*calibrate*/
do_add_lexeme(c2lex19, c2, calibrate, c2cat1, means17, predicate, nil).

/*move*/
do_add_lexeme(c2lex22, c2, move, c2cat1, means20, predicate, nil).

/*go*/
do_add_lexeme(c2lex24, c2, go, c2cat1, means20, predicate, nil).

/*experiment*/
do_add_lexeme(c2lex25, c2, experiment, c2cat1, means21, predicate, nil).

/*---- Nouns ----*/
/*WayStation0*/
do_add_lexeme(c2lex16, c2, waystation0, c2cat2, means14, constant, nil).

/*WayStation1*/
do_add_lexeme(c2lex17, c2, waystation1, c2cat2, means15, constant, nil).

/*WayStation2*/
do_add_lexeme(c2lex18, c2, waystation2, c2cat2, means16, constant, nil).

/*ScienceStation0*/
do_add_lexeme(c2lex20, c2, science0, c2cat2, means18, constant, nil).

/*ScienceStation1*/
do_add_lexeme(c2lex21, c2, science1, c2cat2, means19, constant, nil).
```

```
/*---- Prepositions ----*/
/*see 'at' above.*/

/*to*/
do_add_lexeme(c2lex23, c2, to, c2cat3, nil, nil, c2theta3).
```

6. Where the code lives and how to run it (literally what machine/directory info, svn, instructions on how to run ALFREDJr, demos)