A Program for Translating
English Sentences into
Lisp Programs

Arthur Flatau

AI TR87-48 February 1987
Table of Contents

1. Introduction 1
2. English Grammar 1
3. Features of the English to Lisp translator 2
   3.1. Resolving Anaphoric References 2
   3.2. The Function Knowledge Base 3
      3.2.1. Structure of the Function Knowledge Base 3
      3.2.2. Adding Functions to the Knowledge Base 3
4. How to Run the English to Lisp translator 4
5. Conclusion 5

I. A Sample Run 6
II. The HCPRVR rules and lisp code 10
   II.1. Grammar Rules 10
   II.2. Translation Rules 12
   II.3. Lisp Functions 18
1. Introduction

This report describes a program to translate a set of English sentences into recursive functions in the Lisp programming language. The program takes as input a symbol that will be the name of the function and a set of sentences that the user types in that describe the computation the function should perform. The sentences are then parsed into surface semantic relations, which are then translated into English. The parser and the translator are both implemented as HCPRVR programs. A few top level functions and some other functions that do input and output are written in Lisp. Lisp has much better facilities for input and output that what is provided in HCPRVR.

As an example consider the sentences:

*The sum of a list that is the null list is zero.*

*The sum of a list is the head of the list plus the sum of the tail of the list.*

After reading and parsing these sentence the program will translate them into the following Lisp function. [The program is told that the name of the function is SUM!].

```
(DEFUN SUM! (LIST)
  (COND ((NULL LIST) 0.)
             (T (+ (CAR LIST) (SUM! (CDR LIST))))))
```

The parser of the English to Lisp translator translates the English sentences into surface semantic relations. A surface semantic relation is a list whose first element is the word (more correctly the symbol) which is being described followed by a list of keyword/semantic relation pairs.

2. English Grammar

The subset of English that is understood by the program is very simple. All the sentences consist of a phrase describing what is being computed followed by the verb *is* and finally a phrase describing the result of the computation. The phrase describing what is being computed can include some conditions under which this statement can be used to calculate the result. For example in the sentence:

*The sum of a list that is the null list is zero.*

The description of what is being computed is: *The sum of a list that is the null list* and the result is: *zero*. The description contains a condition: *that is the null list* under which this description can be used. This sentence is only valid when we are considering the sum of a list that has no elements in it.

The English to Lisp translator parser, parses the English sentences into semantic surface relations. It uses fairly traditional linguistic notions of noun phrases, verb phrases, prepositional phrases, etc. However, the grammar is a fairly small because the English that it accepts is very restricted.
3. Features of the English to Lisp translator

The translator works by walking down the semantic surface relation "tree" and translating each component. The component can be a "function", e.g. HEAD in (HEAD "OF (LIST DET THE)). The arguments to the function are the semantic relations of some of the keyword/semantic relations pairs, e.g. (LIST DET THE) is an argument of HEAD since it is the semantic relation of the keyword "OF" in (HEAD "OF (LIST DET THE)). The translator finds the translation of the "function" and then translates each of the arguments of the function. The translator's knowledge base contains the information about which keywords are the arguments of a functions.

3.1. Resolving Anaphoric References

The translator makes use of a very simple technique for resolving anaphoric references. It assumes that a reference to something with a definite article in front of it refers to a "parameter" of the sentence. The parameters of a sentence are the noun phrases of the prepositional phrases of the description of the sentence. For instance, in the sentence The sum of a list ..., the only parameter is a list, in the sentence The removal of an element from a list ... the parameters are a list and an element. The translator prompts the user for the types of the parameters during the translation from semantic surface relations to Lisp.

This information is stored with the definition of the function (see section 3.2.1). Consider the sentence:

The sum of a list is the head of the list plus the sum of the tail of the list.

The translator identifies the occurrence of the list in the head of the list with the argument a list.

If an argument to a function is missing, the translator uses a parameter to the function being defined with the same type. For example in the following sentence:

The sum of a list is the head plus the sum of the tail.

The arguments for HEAD and TAIL are both missing. The translator assumes the missing argument for HEAD and TAIL is a list. In this manner the sentence translates identically to:

The sum of a list is the head of the list plus the sum of the tail of the list.

This translator does not recognize anaphoric pronouns (e.g. it). However it would be straightforward to augment it to consider it a "missing argument". So the following sentences would all be translated identically:

The sum of a list is the head of the list plus the sum of the tail of the list.

The sum of a list is the head it plus the sum of the tail of the list.

The sum of a list is the head it plus the sum of the tail.

The sum of a list is the head it plus the sum of the tail of it.
3.2. The Function Knowledge Base

3.2.1. Structure of the Function Knowledge Base

The function knowledge base is a set of assertions about the different functions that are known to the translator. The translation of a surface semantic relation is done by trying to find an assertion that matches the head of the surface semantic relation. Not all surface semantic relations are translated this way, for instance in the examples in Appendix I the surface semantic relation (LIST DET THE) will fail to match any of the assertions in the function knowledge base. This is because (LIST DET THE) is not a reference to a function, but rather to an "argument" of the function being defined.

An example of a function definition is the following form:

```
((FUNCTION-DEFINITION ELEMENT *NAME CAR
  *ARGS ((*OF LIST))
  *MATCH-FORM (ELEMENT DET THE MOD FIRST)))
```

This is used when trying to translate a surface semantic relation that starts with the symbol ELEMENT. However before using this translation, the surface semantic relation must match the *MATCH-FORM. In other words it must have a determiner (DET) of the and a modifier (MOD) of first. If there is a match then the *NAME keyword pair specifies the function that should be used as the translation, in this case CAR. The *ARGS field specifies the arguments for this function. In this case the surface semantic relation should contain a keyword pair with the keyword *OF and the noun phrase of the *OF should be a list. The last thing in the list is the symbol ELEMENT, this is an HCPRVR marker. When a function is added to the knowledge base, it can replace a previous definition of the function. The translator first removes all axioms that have the name of the function as a marker before translation. This allows the function to be redefined, which is useful not only for debugging the translator, but for developing English descriptions of programs using the translator.

3.2.2. Adding Functions to the Knowledge Base

One of the key features of the English to Lisp translator is that its knowledge base is built up as more and more sentences are translated. Once a function is translated it can then be used in the same way as the "built in" functions of the translator. In the examples from Appendix I a sorting function is defined that makes use of two functions that were defined previously.

Consider the sentence:

The sort of a list is the minimal element of the list concatenated to the sort of the list resulting from the removal of the minimal element of the list.

In this sentence the phrase the minimal element of the list is translated into the surface semantic relations (ELEMENT *OF (LIST DET THE) DET THE MOD MINIMAL). The translator notices that the function MINIMUM! that was given to the translator earlier "matches" this semantic surface relation. Therefore it is translated to (MINIMUM! LIST). The function REMOVE! is also used in the translation of the sorting program in a similar way.

The knowledge base of the translator starts out with a few predefined functions. If this work were to
be extended, presumably there would need to be more predefined functions added. The following table lists the phrases and their translations that are programmed in the knowledge base a priori.

<table>
<thead>
<tr>
<th>English Phrase</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>the head of the list</td>
<td>(CAR LIST)</td>
</tr>
<tr>
<td>the tail of the list</td>
<td>(CDR LIST)</td>
</tr>
<tr>
<td><code>&lt;first operand&gt;</code> concatenated to the list</td>
<td>(CONS <code>&lt;first operand&gt;</code> LIST)</td>
</tr>
<tr>
<td><code>&lt;first operand&gt;</code> plus <code>&lt;second operand&gt;</code></td>
<td>(+ <code>&lt;first operand&gt;</code> <code>&lt;second operand&gt;</code>)</td>
</tr>
<tr>
<td><code>&lt;first operand&gt;</code> is equal to <code>&lt;second operand&gt;</code></td>
<td>(= <code>&lt;first operand&gt;</code> <code>&lt;second operand&gt;</code>)</td>
</tr>
<tr>
<td><code>&lt;first operand&gt;</code> is less than <code>&lt;second operand&gt;</code></td>
<td>(&lt; <code>&lt;first operand&gt;</code> <code>&lt;second operand&gt;</code>)</td>
</tr>
<tr>
<td>the first element of the list</td>
<td>(CAR LIST)</td>
</tr>
</tbody>
</table>

where `<first operand>` and `<second operand>` are some other English phrase that describe that operand to the function, e.g. the head of the list.

4. How to Run the English to Lisp translator

The English to Lisp translator is invoked by calling the function `NL-COMPLIER`. The user is then prompted for the name of the function. The program could figure out the name of the function for itself, e.g. in the example above for computing the sum of the elements in a list, the program could use the name `SUM`. However in the appendix an example of a sorting program is shown, and under this scheme the name of the function would be `SORT`, which is already the name of a Lisp function. By prompting the user for a name of the function this conflict can be avoided. [Of course the user could still use the name `SORT` but then presumably she/he meant to redefined the Lisp function `SORT`.]

After entering the name of the function the user is asked to enter the number of sentences that will be in the definition of the function. Then the user is prompted for and must enter the sentences. Each sentence is parsed before asking for the next one. If the sentence fails to parse the user is asked to enter another sentence to be used instead. Once all the sentences are entered and successfully parsed the program attempts to translate the surface semantic relations generated by parses the sentences into a Lisp program. During the translation the user will be prompted for the types of the arguments of the function being described (see Section 3.1 for a description of what this means). If the translation is successful the program prints out the function that resulted from the parse. If it is unsuccessful it prints out a message that says that the program was unable to do the translation. In both cases the user is then prompted for the name of another function to be translated. When the user becomes tired of typing in all these sentences, she/he can quit by entering `done` as the name of the function to be translated.

There is no way by invoking the function `NL-COMPLIER` to see if the sentences that are entered have multiple parses and/or multiple translations. However it is a straightforward extension to allow this.
5. Conclusion

The program described in this paper translates simple sentences in restricted English to the programming language Lisp. The program has some ability to resolve anaphoric references. In addition, it can build up a knowledge base of functions that have been described in English. Each time a function is defined it is added to the knowledge base and can be used in the definitions of later functions. This allows more and more complex functions to be translated by the program by just slightly augmenting the English language grammer of the program.
I. A Sample Run

The following shows a sample run of the English to Lisp translator, as it translates four sets of sentences into Lisp. In the below dialogue the type in of the user is underlined.

```
(nl-compiler)
Enter the name of the function you wish to define (or DONE if you are done): sum!
Enter the number of sentences there are in the definition of SUM!: 2
Sentence number 1:
Enter a sentence: The sum of a list that is the null list is zero.
The parse is:
(IS AE (SUM *OF (LIST *PROPERTY (LIST DET THE MOD NULL) DET A)
DET THE))
RESULT (ZERO))
Sentence number 2:
Enter a sentence: The sum of a list is the head of the list plus the sum of the tail of the list.
The parse is:
(IS AE
(SUM *OF (LIST DET A) DET THE)
RESULT
(PLUS FIRST-OP (HEAD *OF (LIST DET THE) DET THE)
SECOND-OP (SUM *OF (TAIL *OF (LIST DET THE) DET THE)
DET THE)))
The function SUM! has 1 argument.
The first argument to the function SUM! is: "a list"
Enter the type of "a list": list

The translation is:
(DEFUN SUM! (LIST)
  (COND ((NULL LIST) 0.)
    (T (+ (CAR LIST) (SUM! (CDR LIST))))))
```

Enter the name of the function you wish to define (or DONE if you are done): minimum!
Enter the number of sentences there are in the definition of MINIMUM!: 2
Sentence number 1:
Enter a sentence: The minimal element of a list that has a length of one
Continue: is the first element of the list.
The parse is:
(IS AE (ELEMENT *OF (LIST *PROPERTY (LENGTH *OF (ONE) DET A) DET A)
DET THE
MOD MINIMAL)
RESULT (ELEMENT *OF (LIST DET THE) DET THE MOD FIRST))
Sentence number 2:
Enter a sentence: The minimal element of a list is the first element of
Continue: the list if the first element is less than the minimal element
Continue: of the tail of the list otherwise the result is the minimal
Continue: element of the tail of the list.

The parse is:
(IS AE
  (ELEMENT *OF (LIST DET A) DET THE MOD MINIMAL)
  RESULT
  (ELEMENT *IF (IS-LESS-THAN FIRST-OP (ELEMENT DET THE MOD FIRST)
    SECOND-OP (ELEMENT *OF (TAIL *OF (LIST DET THE)
      DET THE)
    DET THE
    MOD MINIMAL))
  *OTHERWISE (IS AE (RESULT DET THE)
    RESULT (ELEMENT *OF (TAIL *OF (LIST DET THE)
      DET THE)
    DET THE
    MOD MINIMAL))
  *OF (LIST DET THE)
  DET THE
  MOD FIRST))

The function MINIMUM! has 1 argument.
The first argument to the function MINIMUM! is: "a list"
Enter the type of "a list": list

The translation is:
(DEFUN MINIMUM! (LIST)
  (COND ((= (LENGTH LIST) 1.)
    (CAR LIST))
  (T (IF (< (CAR LIST) (MINIMUM! (CDR LIST)))
    (CAR LIST)
    (MINIMUM! (CDR LIST))))))

Enter the name of the function you wish to define (or DONE if you are
done): remove!
Enter the number of sentences there are in the definition of REMOVE!: 2
Sentence number 1:
Enter a sentence: The removal of an element from a list that is
Continue: the null list is the null list.

The parse is:
(IS AE (REMOVAL *OF (ELEMENT DET AN)
  *FROM (LIST *PROPERTY (LIST DET THE MOD NULL) DET A)
  DET THE)
  RESULT (LIST DET THE MOD NULL))
Sentence number 2:
Enter a sentence: The removal of an element from a list is the tail of the
Continue: list if the head of the list is equal to the element, otherwise
Continue: the result is the head of the list concatenated to the removal
Continue: of the element from the tail of the list.

The parse is:
(IS AE (REMOVAL *OF (ELEMENT DET AN)
  *FROM (LIST DET A) DET THE)
  RESULT (TAIL *IF (IS-EQUAL-TO FIRST-OP (HEAD *OF (LIST DET THE)
DET THE)
SECOND-OP (ELEMENT DET THE))
*OTHERWISE (IS AE (RESULT DET THE)
RESULT (CONCATENATED-TO
FIRST-OP (HEAD *OF (LIST DET THE)
DET THE)
SECOND-OP (REMOVAL *OF (ELEMENT DET THE)
*FROM (TAIL
*OF (LIST
DET THE)
DET THE)
DET THE))
*OF (LIST DET THE)
DET THE))

The function REMOVE! has 2 arguments.
The first argument to the function REMOVE! is: "an element"
Enter the type of "an element": number
The second argument to the function REMOVE! is: "a list"
Enter the type of "a list": foo
Enter one of the symbols: NUMBER, LIST, OTHER.
Enter a symbol that is the type of "a list": list

The translation is:
(DEFUN REMOVE! (ELEMENT LIST)
(COND ((NULL LIST) NIL)
       (T (IF (EQUAL (CAR LIST) ELEMENT)
            (CDR LIST)
            (CONS (CAR LIST) (REMOVE! ELEMENT (CDR LIST)))))))

Enter the name of the function you wish to define (or DONE if you are
done): sort!
Enter the number of sentences there are in the definition of SORT!: 2
Sentence number 1:
Enter a sentence: The sort of a list that is the null list is the
Continue: null list.

The parse is:
(IS AE (SORT *OF (LIST *PROPERTY (LIST DET THE MOD NULL) DET A)
DET THE)
RESULT (LIST DET THE MOD NULL))

Sentence number 2:
Enter a sentence: The sort of a list is the minimal element of the list
Continue: concatenated to the sort of the list resulting from the removal
Continue: of the minimal element of the list.

The parse is:
(IS AE (SORT *OF (LIST DET A) DET THE)
RESULT (CONCATENATED-TO FIRST-OP (ELEMENT *OF (LIST DET THE)
DET THE
MOD MINIMAL)
SECOND-OP (SORT *OF
(LIST *RESULT-IN
(REMOVAL *OF

8
(ELEMENT *OF
  (LIST DET
    THE)
  DET THE
    DET THE
  DET THE))

The function SORT! has 1 argument.
The first argument to the function SORT! is: "a list"
Enter the type of "a list": list

The translation is:
(DEFUN SORT (LIST)
  (COND ((NULL LIST) NIL)
    (T (CONS (MINIMUM! LIST) (SORT (REMOVE! (MINIMUM! LIST) LIST))))))

Enter the name of the function you wish to define (or DONE if you are done): done
NIL
II. The HCPRVR rules and Lisp code

II.1. Grammar Rules

The following is a list of the rules used to parse the English sentences in the English to Lisp translator. The function DEF-AXIOMS is a function defined in Section II.3 of this appendix.

;;; All grammar rules have a % as marker so (CLEAR %) will
;;; clear all the rules

(DEF-AXIOMS SNT
  ((SNT _X _Y) % (SNT1 _X _Y NIL))
)

;;; The real sentence rule is SNT1

(DEF-AXIOMS SNT1
  ((SNT1 _X (IS AE _Y1 RESULT _Y2) _R) %
   (NP _X _Y1 (V _X2)) (VERB_V IS) (NP _X2 _Y2 _R))
  ((SNT1 _X (IS AE _Y1 RESULT (Y2 Z4 Y4 Z5 Y5 _Y3)) _R) %
   (NP _X _Y1 (V _X2)) (VERB_V IS) (NP _X2 _Y2 _Y3 _X3)
   (COND-PH_X3 ((Z4 Y4) (Z5 Y5)) _R))
  ((SNT1 _X (IS AE _Y1 RESULT _Y2) _R) %
   (NP _X _Y1 (V _X2)) (VERB_V IS) (NP _X2 _Y2 _Y2 _R))
)

(DEF-AXIOMS SNT2
  ((SNT2 _X (_Y2 FIRST-OP _Y1 SECOND-OP _Y3) _R) %
   (NP _X _Y1 _X2) (VP _X2 (_Y2 _Y3) _R))
)

(DEF-AXIOMS NP
  ((NP _X _Y1 _R) % (NP1 _X1 _Y1 _R))
  ((NP _X _Y1 _Y3 _Y4 _Y2 _R) %
   (NP1 _X1 (_Y1 _Y2) _X2) (POSS-PH_X2 (_Y3 _Y4) _R))
  ((NP _X _Y1 (_Y1 _Y3 _Y4 _Y5 _Y6 _Y2) _R) %
   (NP1 _X1 (_Y1 _Y2) _X2) (CONJ-PH_X2 (_Y3 _Y4 _Y5 _Y6) _R))
  ((NP _X _Y1 _Y3 _Y4 _Y2 _R) %
   (NP1 _X1 (_Y1 _Y2) _X2) (CONJ-PH_X2 (_Y3 _Y4) _R))
  ((NP _X _Y1 _Y3 _Y4 _Y2 _R) %
   (NP1 _X1 (_Y1 _Y2) _X2) (VMOD_X2 (_Y3 _Y4) _R))
)

(DEF-AXIOMS NP1
  ((NP1 (_X1 _X2 _X3) (_Y2 DET _Y1) _X3) %
   (ART X1 _X1) (NOUN _X2 _Y2))
  ((NP1 (_X1 _X2) (_Y1) _X2) %
   (NOUN _X1 _Y1))
  ((NP1 (_X1 _X2 _X3 _X4) (_Y3 DET _Y1 MOD _Y2) _X4) %
   (ART X1 _Y1) (ADJ _X2 _Y2) (NOUN _X3 _Y3))
)

(DEF-AXIOMS VP
  ((VP _X1 (_Y1 _Y2) _R) %
   (VP1 _X1 _Y1 _X2) (NP _X2 _Y2 _R))
)

(DEF-AXIOMS VP1
  ((VP1 (_X1 _X2) _Y1 _X2) % (VERB _X1 _Y1))
  ((VP1 (_V _X1 _X2 _X3) _Y3 _X3)
    (VERB_V IS) (VERB _X1 _Y1) (AUXILIARY _X2 _Y2) (SEP _Y1 _Y2 _Y3))
  ((VP1 (_X1 _X2 _X3) _Y3 _X3)
    (VERB _X1 _Y1) (AUXILIARY _X2 _Y2) (SEP _Y1 _Y2 _Y3))
)
(DEF-AXIIOMS VMOD
  ((VMOD X1 (*RESULT-IN Y2)_R) %
   (VP1 X1 *RESULT-IN X2) (NP X2 Y2 R)))

(DEF-AXIIOMS POSS-PH
  ((POSS-PH (X1 . X2) (Y1 Y2) R) %
   (POSS X1 Y1) (NP X2 Y2 R)))

(DEF-AXIIOMS CONJ-PH
  ((CONJ-PH (X1 X2 . X3) (Y3 Y4) R) %
   (CONJUNCTION X1 Y1) (VERB X2 Y2) (SEF Y1 Y2 Y3) (NP X3 Y4 R)))

(DEF-AXIIOMS COND-PH
  ((COND-PH (X2 X1) ((IF Y1) (OTHERWISE Y2)) R) %
   (COND2 X1 Y1 (OTHERWISE . X2) (COND1 X2 Y2 R)))

(DEF-AXIIOMS CONN-POSS-PH
  ((CONN-POSS-PH X1 (Y1 Y2 X3 Y4) R) %
   (POSS-PH X1 (Y1 X2) (Y2 . X3)) (CONNECTOR X2 Y3) (NP X3 Y4 R)))

;;; Lexicon: All rules in the lexicon have a $ as marker so (CLEAR $)
;;; will clear the entire lexicon

;; Nouns
(DEF-AXIIOMS NOUN
  ((NOUN X Y) % (X NOUN Y)))

(DEF-AXIIOMS NOUNS
  ((SUM NOUN SUM) $)
  ((LIST NOUN LIST) $)
  ((HEAD NOUN HEAD) $)
  ((TAIL NOUN TAIL) $)
  ((SORT NOUN SORT) $)
  ((ZERO NOUN ZERO) $)
  ((ONE NOUN ONE) $)
  ((ELEMENT NOUN ELEMENT) $)
  ((LENGTH NOUN LENGTH) $)
  ((RESULT NOUN RESULT) $)
  ((REMOVAL NOUN REMOVAL) $))

;; Verbs
(DEF-AXIIOMS VERB
  ((VERB X Y) % (X VERB Y)))

(DEF-AXIIOMS VERBS
  ((IS VERB IS) $)
  ((HAS VERB HAS) $)
  ((PLUS VERB PLUS) $)
  ((CONCATENATED VERB CONCATENATED) $)
  ((EQUAL VERB EQUAL) $)
  ((LESS VERB LESS) $)
  ((RESULTING VERB RESULTING) $))

(DEF-AXIIOMS AUXILIARY
  ((AUXILIARY THAN THAN) $)
((AUXILIARY TO TO) $)
((AUXILIARY FROM FROM) $))

;; Adjectives
(DEF-AXIOMS ADJ
  ((ADJ NULL NULL) $)
  ((ADJ MINIMAL MINIMAL) $)
  ((ADJ FIRST FIRST) $))

;; Conjunctions
(DEF-AXIOMS CONJUNCTION
  ((CONJUNCTION THAT THAT) $))

;; Possessives
(DEF-AXIOMS POSS
  ((POSS OF *OF) $)
  ((POSS WITH *WITH) $))

;; Articles
(DEF-AXIOMS ART
  ((ART THE THE) $)
  ((ART AN AN) $)
  ((ART A A) $))

(DEF-AXIOMS CONNECTOR
  ((CONNECTOR FROM *FROM) $))

;;; Semenatic Event Forms
(DEF-AXIOMS SEF
  ((SEF EQUAL TO IS-EQUAL-TO) $)
  ((SEF LESS THAN IS-LESS-TAN) $)
  ((SEF CONCATENATED TO CONCATENATED-TO) $)
  ((SEF THAT IS *PROPERTY) $)
  ((SEF THAT HAS *PROPERTY) $)
  ((SEF RESULTING FROM *RESULT-IN) $))

II.2. Translation Rules

The following is a list of the rules used to translate the parsed English sentences into Lisp. The function DEF-AXIOMS is a function defined in Section II.3 of this appendix.

;;; Here is where we convert the intermediate semantic form to
;;; the Lisp form we desire.

(DEF-AXIOMS GET-PROP
  ((GET-PROP _PROP (_SYM _LIST-OF-PAIRS) _VAL) +
   (GET-PAIR-VALUE _PROP _LIST-OF-PAIRS _VAL) !))

(DEF-AXIOMS GET-PAIR-VALUE
  ((GET-PAIR-VALUE _PROP NIL NIL) +)
  ((GET-PAIR-VALUE _PROP (_PROP _VAL _LIST-OF-PAIRS) _VAL) +)
  ((GET-PAIR-VALUE _PROP (PROP _VAL _LIST-OF-PAIRS) _VAL) +
    (NEQ _PROP _PROP1) ! (GET-PAIR-VALUE _PROP _LIST-OF-PAIRS _VAL) )
)
(DEF-AXIOMS PUT-PROP
  ((PUT-PROP_PROP_VAL (_SYM _LIST-OF-PAIRS) (_SYM _NEW-PROPS)) +
   (PUT-PAIR-VALUE_PROP_VAL _LIST-OF-PAIRS _NEW-PROPS))
)

(DEF-AXIOMS PUT-PAIR-VALUE
  ((PUT-PAIR-VALUE_PROP_VAL NIL (_PROP _VAL)) +)
  ((PUT-PAIR-VALUE_PROP_VAL (_PROP _VAL1 _LIST-OF-PAIRS)
    (_PROP _VAL _LIST-OF-PAIRS)) +
   (PUT-PAIR-VALUE_PROP_VAL (_PROP1 _VAL1 _LIST-OF-PAIRS)
    (_PROP1 _VAL1 _NEW-PAIRS)) +
   (NEQ* _PROP _PROP1) !
   (PUT-PAIR-VALUE_PROP_VAL _LIST-OF-PAIRS _NEW-PAIRS)))

;;; The determined determiners.
(DEF-AXIOMS DETERMINER
  ((DETERMINER THE) +)
  ((DETERMINER A) +)
  ((DETERMINER AN) +))

(DEF-AXIOMS NON-DETERMINED
  ((NON-DETERMINED A) +)
  ((NON-DETERMINED AN) +))

;;; SUBSET-EQ Succeeds if arg1 is the same as arg2, except arg2 can
;;; have extra properities in its property list
(DEF-AXIOMS SUBSET-EQ
  ((SUBSET-EQ (_SYM) (_SYM _THING)) +)
  ((SUBSET-EQ (_SYM PROP _VAL _PAIRS1) (_SYM _PAIRS2)) +
   (GET-PAIR-VALUE_PROP_PAIRS2 _VAL)
   (SUBSET-EQ (_SYM _PAIRS1) (_SYM _PAIRS2))))

;;; GET-ARGS returns the arguments from a list of triples which has
;;; a first element something like (LIST DET A ...) or
;;; (REMOVAL *OF (ELEMENT *FROM (LIST DET A ...) DET A ...)) DET THE ...
;;; the arguments returned for these cases are (LIST) and (ELEMENT LIST)
;;; respectively. It also adds assertions of the form:
;;; (FIND-ARG _THING LIST) <FIND-ARG
;;; (DETERMINED _X) (SUBSET-EQ (LIST DET _X) _THING))
;;; in the first case and two similar assertions in the second case.
;;; Then doing a (CLEAR <FIND-ARG) will clear these.
(DEF-AXIOMS GET-ARGS
  ((GET-ARGS _TRIPLE _TRIPLES) _TOP-NAME _V _ARG-PAIRS) +
  (GET-ARGS-INTERNAL _TRIPLE _ARGS _U) (CHECK-ARGS _TRIPLES _ARGS)
  (CLEAR <FIND-ARG) (CLEAR <DEFAULT-ARG)
  (SETV* _V (GET-ARG-TYPES _TOP-NAME _ARGS))
  (MERGE-INTO-PAIRS _U _V _ARG-PAIRS)
  (ENTER-ARG-ASSERTIONS _V))

(DEF-AXIOMS GET-ARGS-INTERNAL
  ((GET-ARGS-INTERNAL ((_SYM _PROPS) _REST) ((_SYM (_SYM DET _X)))
    (*OF)) +
   (GET-PAIR-VALUE DET _PROPS X) (NON-DETERMINED _X))
  ((GET-ARGS-INTERNAL ((_SYM1 _PROPS1) (_SYM2 _PROPS2)) _REST)
    ((_SYM1 (_SYM1 DET _X)) (_SYM2 (_SYM2 DET _Y)))
    (*OF *FROM)) +
  (GET-PAIR-VALUE DET _PROPS1 _X) (NON-DETERMINED _X))
((GET-PAIR-VALUE DET _PROPS2 Y) (NON-DETERMINED X)))

(DEF-AXIOMS MERGE-INTO-PAIRS
  ((MERGE-INTO-PAIRS NIL NIL NIL) *)
  ((MERGE-INTO-PAIRS (_X1 , _X2) (_Y1 . _Y2) ((_X1 _Y1) , _Z)) *)
  ((MERGE-INTO-PAIRS _X2 , _X2 , _Y2 , Z)))

(DEF-AXIOMS CHECK-ARGS
  ((CHECK-ARGS NIL _ARGS) *)
  ((CHECK-ARGS _TRIPLE , _TRIPLES ARG) *)
  ((GET-ARGS-INTERNAL _TRIPLE ARG _IGNORE) (CHECK-ARGS _TRIPLES ARG))

(DEF-AXIOMS FIND-AN-ARG
  ((FIND-AN-ARG (_ARG , _ARGS) _TYPE _ACTUAL) *)
  ((FIND-AN-ARG _ARG _ARGS _ACTUAL) (DEFAULT-ARG _ACTUAL _TYPE))
  ((FIND-AN-ARG (_ARG , _ARGS) _TYPE _ACTUAL) *)
  ((FIND-AN-ARG _ARG _ARGS _TYPE _ACTUAL))
  
  ;;; Converts a triple of the form (arg condition result) like
  ;;; above to (NULL LIST) or T

(DEF-AXIOMS CONVERT-CONDITION
  ;; The SETV* below is bogus, but putting just T doesn’t
  ;; seem to work (because T is a variable).
  ((CONVERT-CONDITION (_ARGS NIL RES) _V) (+ (SETV* _V (TRUE)))
  ((CONVERT-CONDITION (_ARGS _COND RES) (NULL _ACTUAL-ARG)) *)
  ((FIND-AN-ARG _ARGS LIST _ACTUAL-ARG)
   (SUBSET-EQ (LIST DET THE MOD NULL _COND))
  ((CONVERT-CONDITION (_ARGS _COND RES)
   (= (LENGTH _ACTUAL-ARG) _ACTUAL-CODE)) *)
  ((FIND-AN-ARG _ARGS LIST _ACTUAL-ARG)
   (SUBSET-EQ (LENGTH DET A) _COND)
   (GET-PROP *OF _COND _CODE)
   (CONVERT-CODE _CODE _ACTUAL-CODE)))

(DEF-AXIOMS CONVERT-CONSTANT
  ((CONVERT-CONSTANT (ZERO) 0) *)
  ((CONVERT-CONSTANT (ONE) 1) *)
  ((CONVERT-CONSTANT _FORM NIL) *)
  ((SUBSET-EQ (LIST DET THE MOD NULL _FORM)))

  ;;; Returns a list with the function as the head of the list,
  ;;; and place holders for the arguments doing a get with each
  ;;; placeholder on the args (from the parse listed) and converting
  ;;; that should provide the actual arguments

(DEF-AXIOMS FUNCTION-DEFINITION
  ((FUNCTION-DEFINITION HEAD *NAME CAR
    *ARGS ((*OF LIST))
    *MATCH-FORM (HEAD)) HEAD)

  ((FUNCTION-DEFINITION TAIL *NAME CDR
    *ARGS ((*OF LIST))
    *MATCH-FORM (TAIL)) TAIL)

  ((FUNCTION-DEFINITION CONCATENATED-TO *NAME CONS
    *ARGS ((FIRST-OP NUMBER) (SECOND-OP LIST))
    *MATCH-FORM (CONCATENATED-TO) CONCATENATED-TO)

  ((FUNCTION-DEFINITION PLUS *NAME +
    *ARGS ((FIRST-OP NUMBER) (SECOND-OP LIST))
    *MATCH-FORM (PLUS)) PLUS)
((FUNCTION-DEFINITION IS-EQUAL-TO *NAME EQUAL
*ARGS ((FIRST-OP NUMBER) (SECOND-OP LIST))
*MATCH-FORM (IS-EQUAL-TO) IS-EQUAL-TO))

((FUNCTION-DEFINITION IS-LESS-THAN *NAME <
*ARGS ((FIRST-OP NUMBER) (SECOND-OP LIST))
*MATCH-FORM (IS-LESS-THAN) IS-LESS-THAN)

((FUNCTION-DEFINITION ELEMENT *NAME CAR
*ARGS (*OP LIST)
*MATCH-FORM (ELEMENT DET THE MOD FIRST) ELEMENT))

(DEF-AXIOMS FIND-FUNCTION
 ((FIND-FUNCTION (FN . _ARGS) (_FN . _PAIRS)) +
 (FUNCTION-DEFINITION FN . _PAIRS)
 (GET-PAIR-VALUE *MATCH-FORM _PAIRS _MF)
 (SUBSET-EQ _MF (_FN . _ARGS)))

(DEF-AXIOMS GET-ONLY-ARGS
 ((GET-ONLY-ARGS NIL NIL) +)
 ((GET-ONLY-ARGS (_X . _Y) _Z) (_X . _Z1) +
 (GET-ONLY-ARGS _Z _Z1))

(DEF-AXIOMS CONVERT-FUNCTION
 ((CONVERT-FUNCTION _FORM (_NAME . _ARGS-TYPES)) +
 (FUNCTION-DEFINITION _FORM _PLIST)
 (GET-PROP *NAME _PLIST _NAME)
 (GET-PROP *ARGS _PLIST _ARGS-TYPES)))

(DEF-AXIOMS GET-ARGS-TYPES
 ((GET-ARGS-TYPES _FORM _ARGS-TYPES) +
 (FUNCTION-DEFINITION _FORM _PLIST) (GET-PROP *ARGS _PLIST _ARGS-TYPES)))

(DEF-AXIOMS CONVERT-AN-ARG-INTERNAL
 ((CONVERT-AN-ARG-INTERNAL (_FN . UNCONVERT-ARGS)
   (_REAL-FN . _REAL-ARGS)) +
 (FUNCTION-DEFINITION (_FN . UNCONVERT-ARGS) (_REAL-FN . _TEMPLATE))
 (CONVERT-ARGS TEMPLATE UNCONVERT-ARGS _REAL-ARGS))
 ((CONVERT-AN-ARG-INTERNAL UNCONVERT-ARG_REAL-ARG) +
 (GET-PROP *RESULT-IN UNCONVERT-ARG NIL)
 (FIND-ARG UNCONVERT-ARG_REAL-ARG))
 ((CONVERT-AN-ARG-INTERNAL UNCONVERT-ARG_REAL-CODE) +
 (GET-PROP *RESULT-IN UNCONVERT-ARG _STUFF)
 (CONVERT-CODE _STUFF REAL-CODE))
 ((CONVERT-AN-ARG-INTERNAL UNCONVERT-ARG_REAL-ARG) +
 (GET-DEFAULT-ARG UNCONVERT-ARG_REAL-ARG))

(DEF-AXIOMS CONVERT-AN-ARG
 ((CONVERT-AN-ARG_ARG-NAME ARG-TYPE _ARGS REAL-ARG) +
 (GET-PAIR-VALUE ARG-NAME _ARGS UNCONVERTED-ARG)
 (NEG* _UNCONVERTED-ARG NIL) !
 (CONVERT-AN-ARG-INTERNAL UNCONVERTED-ARG_REAL-ARG))
 ((CONVERT-AN-ARG_ARG-NAME ARG-TYPE _ARGS_REAL-ARG) +
 (GET-PAIR-VALUE ARG-NAME _ARGS NIL) (DEFAULT-ARG_ARG-TYPE _REAL-ARG)))

(DEF-AXIOMS CONVERT-ARGS
 ((CONVERT-ARGS NIL _ARGS NIL) +
 (CONVERT-ARGS ((ARG-NAME ARG-TYPE) _REST) _ARGS
 (_REAL-ARG _REAL-ARGS)) +
(CONVERT-AN-ARG_ARG-NAME_ARG-TYPEARGS_REAL-ARG)
(CONVERT-ARGS_RESTARGS_REAL-ARGS))

;;; An else branch should be a form like
;;; (IS AE (RESULT DET THE) RESULT ...), we just need
;;; to convert the ...
(DEF-AXIOMS CONVERT-ELSE
  ((CONVERT-ELSE (IS . _PAIRS) _CODE) +
    (GET-PAIR-VALUE AE_PAIRS (RESULT DET THE))
    (GET-PAIR-VALUE RESULT_PAIRS _STUFF)
    (CONVERT-CODE _STUFF _CODE)))

(DEF-AXIOMS CONVERT-CODE
  ((CONVERT-CODE (_RES-FN . _RES-ARGS) (_FN . _REAL-FORM)) +
    (GET-PAIR-VALUE *IF_RES-ARGS NIL)
    (CONVERT-FUNCTION (_RES-FN . _RES-ARGS) (_FN . _TEMPLATE))
    (CONVERT-ARGS_TEMPLATE_RES-ARGS_REAL-FORM))
  ((CONVERT-CODE (_RES-FN . _RES-ARGS)
    (IF_COND-CODE (_RES-FN . _REAL-FORM) ELSE-CODE)) +
    (CONVERT-FUNCTION (_RES-FN . _RES-ARGS) (_FN . _TEMPLATE))
    (CONVERT-ARGS_TEMPLATE_RES-ARGS_REAL-FORM)
    (GET-PAIR-VALUE *IF_RES-ARGS_COND) (NEQ* _COND NIL) !
    (CONVERT-CODE_COND_COND-CODE)
    (GET-PAIR-VALUE *OTHERWISE_RES-ARGS_ELSE)
    (CONVERT-ELSE_ELSE_ELSE-CODE))
  ((CONVERT-CODE_RES_CONST) +
    (CONVERT-CONSTANT_RES_CONST)))

(DEF-AXIOMS MAKE-COND-INTERNAL
  ((MAKE-COND-INTERNAL NIL NIL) +)
  ((MAKE-COND-INTERNAL ((ARG_COND_RES) . TRIPLES)
    ((RES-COND_RES-CODE) . REST)) +
    (CONVERT-CONDITION_ARG_COND_RES) _RES-COND)
    (CONVERT-CODE_RES_RES-CODE)
    (MAKE-COND-INTERNAL_TRIPLES_REST)))

(DEF-AXIOMS MAKE-COND
  ((MAKE-COND_TRIPLES (COND . RES-TRIPLES)) +
    (MAKE-COND-INTERNAL_TRIPLES_RES-TRIPLES))

(DEF-AXIOMS ENTER-ARG-ASSERTIONS
  ((ENTER-ARG-ASSERTIONS NIL) +)
  ((ENTER-ARG-ASSERTIONS (ARG . _ARGS)) +
    (ASSERT (FIND-ARG_THING_ARG) <FIND-ARG
      (DETERMINER_X)
      (SUBSET-EQ (ARG DET X) _THING))
      (ENTER-ARG-ASSERTIONS_ARGS)))

(DEF-AXIOMS MAKE-TRIPLE
  ((MAKE-TRIPLE_A_B_C (A_B_C) +))

;;; Returns a list of triples of the form (<arg> <condition> <result>)
;;; where <arg> is something like (LIST DET A), with perhaps some
;;; extra properties in it, <condition> is something like
;;; (LIST DET THE MOD NULL) or NIL and result is something like (ZERO)
;;; or (PLUS FIRST-OF ... SECOND-OF ...)
(DEF-AXIOMS MAKE-COND-RESULT-TRIPLE
((MAKE-COND-RESULT-TRIPLE NIL NIL NIL) +)
((MAKE-COND-RESULT-TRIPLE ((_FIRST _SECOND) . _FORMS)
  (_TRIPLE . _RESULTS)) +)
(GET-PROP *OF _FIRST _ARG)
(GET-PROP *PROPERTY _ARG _COND)
(GET-PROP *FROM _FIRST NIL)
(MAKE-TRIPLE (_ARG) _COND _SECOND _TRIPLE)
(MAKE-COND-RESULT-TRIPLE _FORMS _RESULTS))
((MAKE-COND-RESULT-TRIPLE ((_FIRST _SECOND) . _FORMS)
  (_TRIPLE . _RESULTS)) +)
(GET-PROP *OF _FIRST _ARG)
(GET-PROP *PROPERTY _ARG NIL)
(GET-PROP *FROM _FIRST _F)
(NEQ* _F NIL) !
(GET-PROP *PROPERTY _COND)
(MAKE-TRIPLE (_ARG _F) _COND _SECOND _TRIPLE)
(MAKE-COND-RESULT-TRIPLE _FORMS _RESULTS)))

(DEF-AXIOMS MERGE-SENTS
 ((MERGE-SENTS NIL NIL NIL) +)
 ((MERGE-SENTS (_SENT1 . _SENTS) (_COND1 _RES1) . _RESULTS)) +
 (GET-PROP ARE _SENT1 _COND1)
 (GET-PROP RESULT _SENT1 _RES1)
 (MERGE-SENTS _SENTS _RESULTS)))

(DEF-AXIOMS GET-FUNCTION-DEF
 ((GET-FUNCTION-DEF ((_NAME . _PROP) . _REST) _NAME) +))

(DEF-AXIOMS GET-MATCH-FORM
 ((GET-MATCH-FORM ((_NAME . _PROP) . _REST) (_NAME DET _X)) +
  (GET-PAIR-VALUE DET _PROPS _X)
  (GET-PAIR-VALUE MOD _PROPS NIL))
 ((GET-MATCH-FORM ((_NAME . _PROP) . _REST) (_NAME DET _X MOD _Y)) +
  (GET-PAIR-VALUE DET _PROPS _X)
  (GET-PAIR-VALUE MOD _PROPS _Y)
  (NEQ* _Y NIL) !))

(DEF-AXIOMS ENTER-FUNCTION
 ((ENTER-FUNCTION _TOP-NAME (_MERGE-SENT . _IGNORE) _ARGS) +
  (GET-FUNCTION-DEF _MERGE-SENT _DEF)
  (GET-MATCH-FORM _MERGE-SENT _FORM)
  (ASSERT (FUNCTION-DEFINITION _DEF *NAME _TOP-NAME
    *ARGS _ARGS
    ;; The below allows (CLEAR FOO) to
    ;; clear the definition of FOO
    *MATCH-FORM _FORM) _TOP-NAME)))

(DEF-AXIOMS CONVERT-TOP
 ((CONVERT-TOP _SENTS (DEFUN _TOP-NAME _ACTUAL-ARGS _COND-FORM) _TOP-NAME) +
  (CLEAR _TOP-NAME)
  (MERGE-SENTS _SENTS _MERGE-SENTS)
  (MAKE-COND-RESULT-TRIPLE _MERGE-SENTS _TRIPLES)
  (GET-ARGS _TRIPLES _TOP-NAME _ACTUAL-ARGS _ARGS-TO-ENTER)
  (ENTER-FUNCTION _TOP-NAME _MERGE-SENTS _ARGS-TO-ENTER)
  (MAKE-COND _TRIPLES _COND-FORM))))
II.3. Lisp Functions

The following Lisp functions are used in the English to Lisp Translator:

;;; This file contains all the lisp functions for the
;;; natural language compiler

;;; First a macro for defining axioms
(DEFVAR ALL-RULE-NAME NIL)

(DEFMACRO DEF-AXIOMS (NAME &BODY AXIOMS)
 ' (PROGN (REMPROP ',NAME 'AXIOMS)
            (SETQ ALL-RULE-NAME (UNION (LIST ',NAME) ALL-RULE-NAME))
            (AXIOMS ',AXIOMS)
            ',NAME))

;;; Make clear usable in HCPRVR
(PUTPROP 'CLEAR T 'FN)

;;; The following functions are for prompting the user for input

(DEFUN GET-A-LINE (PROMPT)
 "Prompts the user and returns the string read."
 (PRINC PROMPT)
 (DO ((LINE (STRING-TRIM '(#\Space #\Tab) (READLINE))
                 (STRING-TRIM '(#\Space #\Tab) (READLINE))))
       (NOT (= (STRING-LENGTH LINE) 0)) LINE)
 (BEEP)
 (TERPRI)
 (PRINC "Enter some text fool!")
 (TERPRI)
 (PRINC PROMPT))

(DEFUN MAKE-STRING-INTO-LIST (STRING)
 (LET* ((LAST-CHAR (AREF STRING (\- (STRING-LENGTH STRING) 1))))
        (END-SENTENCE (COND ((CHAR= LAST-CHAR #\. ) :PERIOD)
                              (T NIL)))
        (START 0)
        ;; This are equivalent to spaces.
        (STRING-CHARS '(#\Space #\, #\: #\Return #\Line #\Tab))
        L)
        (COND ((NOT (NULL END-SENTENCE))
               (SETQ STRING
               (STRING-APPEND (SUBSTRING STRING
                                0
                                (- (STRING-LENGTH STRING) 1))
                                " ")))
               (T (SETQ STRING (STRING-APPEND STRING " "))))
    (DOTTIMES (I (STRING-LENGTH STRING))
        (COND ((MEM #\'CHAR= (AREF STRING I) STRING-CHARS)
               (COND ((= I START)
                       ;; Then several spaces in a row
                       (SETQ START (+ START 1)))
               (T
                (PUSH (INTERN (STRING-UPCASE (SUBSTRING STRING
                                               START I)))
                L))))
Translating English Sentences into Lisp Programs

(SETQ START (+ I 1)))
)
)(VALUES (NREVERSE L) END-SENTENCE))

(DEFUN READ-SENTENCE NIL
"Returns the sentence read as a list of symbols."
(TERPRI)
(LET ((PROMPT "Enter a sentence: " "Continue: ")
      END-OF-SENTENCE BUF)
   (AND END-OF-SENTENCE (NOT (NULL SENTENCE))
      SENTENCE)
   (SETQ BUF (GET-A-LINE PROMPT))
   (MULTIPLE-VALUE-BIND (L KIND)
      (MAKE-STRING-INTO-LIST BUF)
      (COND ((EQ KIND :PERIOD)
             (SETQ END-OF-SENTENCE T))
             (SETQ SENTENCE (APPEND SENTENCE L)))
             (T (SETQ SENTENCE (APPEND SENTENCE L)))))))

(DEFUN GET-TYPES NIL
"A list of types that are known so far."

(DEFUN GET-SYMBOL ()
 (MULTIPLE-VALUE-BIND (SYM KIND)
  (READ)
  (DO ()
    ((EQ KIND 'SYMBOL) SYM)
      (FORMAT STANDARD-OUTPUT " ~Enter a symbol please: ")
      (MULTIPLE-VALUE (SYM KIND) (READ)))))

(DEFUN PRINT-ARG (ARG)
  (LET ( ((DET (GET (SECOND ARG) 'DET)))
     (TYO #" STANDARD-OUTPUT)
     (WHEN (NOT (NULL DET))
       (PRINC (STRING-UPCASE DET) STANDARD-OUTPUT)
       (PRINC " " STANDARD-OUTPUT)
       (PRINC (STRING-UPCASE (FIRST (SECOND ARG))) STANDARD-OUTPUT)
       (TYO #" STANDARD-OUTPUT)))

(DEFUN GET-ARG-TYPES (FUN-NAME ARGS)
  "Prompts the user for the type of the arguments (ARGS) to the function
  FUN-NAME"
  (FORMAT T "-2%The function ~A has ~D argument~P."~A"
    FUN-NAME (LENGTH ARGS))
  (DO ( (I 1 (1+ I))
        (AS ARGS (CDR AS))
        (ACTUAL-ARGS)
        ((NULL AS) ACTUAL-ARGS)
        (SETQ ACTUAL-ARGS (NCONC ACTUAL-ARGS (LIST (FIRST (FIRST AS))))))
      (FORMAT STANDARD-OUTPUT "-4The ")
      (LET ((FORMAT:*FORMAT-OUTPUT* STANDARD-OUTPUT))
         (FORMAT:ONUM I :ORDINAL))
      (FORMAT STANDARD-OUTPUT " argument to the function ~A is: " FUN-NAME)
      (PRINT-ARG (CAR AS))
      (PRINT-ARG (CAR AS))
      (PRINC ": ")

19
(LET ((SYM (DO ((SYM (GET-SYMBOL) (GET-SYMBOL))
 (OR (MEMQ SYM KNOWN-TYPES)
 (EQ SYM 'OTHER)) SYM)
 (FORMAT STANDARD-OUTPUT "Enter one of the symbols: ")
 (FORMAT:PRINT-LIST STANDARD-OUTPUT "A"
 (APPEND KNOWN-TYPES (LIST 'OTHER)))
 (FORMAT STANDARD-OUTPUT ".")
 (FORMAT STANDARD-OUTPUT
 "Enter a symbol that is the type of ")
 (PRINT-ARG (CAR AS)) (PRINC ": ")))
 (WHEN (EQ SYM 'OTHER)
 (PROG NIL
 TOP
 (FORMAT STANDARD-OUTPUT "Enter a new symbol that is the type ")
 (PRINT-ARG (CAR AS)) (PRINC ": ")
 (SETQ SYM (GET-SYMBOL))
 (WHEN (AND (NOT (MEMQ SYM KNOWN-TYPES))
 (NEQ SYM 'OTHER))
 (PUSH SYM KNOWN-TYPES)
 (RETURN T))))
 (AXIOMS `(((DEFAULT-ARG ,SYM , (FIRST (FIRST AS)) <DEFAULT-ARG)))))

(PUTPROP 'GET-ARGS-TYPES T 'FN)

(DEFMACRO GET-PARSE NIL
 '(THIRD (FIRST (FIRST VAL)))))

(DEFUN TOP-LEVEL (SYM)
 (LET ((QFLAG NIL)
 (LFLAG T)
 SENTENCES PARESES)
 (DOTIMES (I
 (PROMPT-AND-READ
 :NUMBER
 "Enter the number of sentences there are in ~
 the definition of ~A: "
 SYM))
 (FORMAT STANDARD-OUTPUT "Sentence number ~D:" (+ I 1))
 (LET ((SENTENCE (READ-SENTENCE)))
 (TRY (LIST 'SNT SENTENCE 'X))
 (DO ()
 ((NOT (NULL VAL)))
 (FORMAT STANDARD-OUTPUT
 "The sentence ~A failed to parse." SENTENCE)
 (FORMAT STANDARD-OUTPUT
 "Enter a new sentence for sentence number ~D:"
 (+ I 1))
 (SETQ SENTENCE (READ-SENTENCE))
 (TRY (LIST 'SNT SENTENCE 'X))
 (SETQ SENTENCES (NCONC SENTENCES (LIST SENTENCE)))
 (FORMAT STANDARD-OUTPUT ".2~A failed to translate."
 (GRIND-TOPL-LEVEL (GET-PARSE))
 (SETQ PARESES (NCONC PARESES (LIST (GET-PARSE))))
 (TRY (LIST 'CONVERT-TOPL PARESES 'X SYM))
 (COND ((NULL VAL)
 (FORMAT STANDARD-OUTPUT
 "The sentences for ~A failed to translate."
 1))))

20
SYM
(T (FORMAT STANDARD-OUTPUT "-3%The translation is:")
(GRIND-TOP-LEVEL (GET-PARSE))))

(DEFUN GET-A-SYM NIL
  (FORMAT T
    "-%Enter the name of the function you wish to define ~
      (or DONE if you are done): ")
  (GET-SYMBOL))

(DEFUN NL-COMPILER NIL
  (DO ((SYM (GET-A-SYM) (GET-A-SYM)))
    ((EQ SYM 'DONE))
    (TOP-LEVEL SYM)
    (TERPRI) (TERPRI) (TERPRI)))