An Approach to Natural Language Parsing

Jack Park

Browsnville, CA

Abstract

A description of the design of an expectation-based parser is given. The parser uses a three-part structure consisting of primitive procedures, expectation procedures and heuristics, and a lexical dictionary. By use of this structure in a Forth environment, a self-parsing vocabulary is built.

Introduction

The development of a Forth-based natural language parser as part of a larger project has uncovered some interesting structural similarities in the parser and Forth. A discussion of the design of that parser using a simple sentence to parse permits a description of the similarities. Powerful language parsers based upon the use of knowledge about the domain in which the universe of discourse resides have been built by artificial intelligence researchers. One such parser, Dyer [DYE83] has been used as a model on which to build the parser discussed here. Throughout this paper, the descriptions of elements of the parser being discussed will follow the descriptions used in [DYE83] since that reference is a useful resource in extension of this work.

Language parsing may be viewed as a process comparable to compiling in a Forth environment but with a few differences worth considering. Sentences in Forth begin with a colon and end with a semicolon. But sentences in a natural language (a language used by humans to convey meaning in domains other than instructions to a computer) do not begin and end with so definite a set of signs. Further, sentences in a natural language may be based on a vocabulary of words far larger than those required to execute any Forth language program. Thus, natural language parsing may be considered a process similar to Forth compiling, but dealing with a far wider range of issues.

It is the process of dealing with the wider range of issues in a natural language domain that requires extensions to the Forth environment. It is not enough simply to add a group of words to the Forth dictionary; methods of dealing with certain parsing issues must also be added. These methods include lexical additions to the dictionary, and domain specific knowledge in the form of procedures or heuristic rules designed to solve parsing problems as they arise. A third addition is a set of primitive meaning structures on which the definitions in the lexicon will be based.

Perhaps the primary way in which domain knowledge is required is for the purpose of determining appropriate word sense or meaning when ambiguity exists. An example of such ambiguity is given with the following sentence which is used later in the development of a parser design. The sentence:

```
BILL PICKED UP THE HAMMER.
```

contains the ambiguous independent word `PICKED`. A first cut shows two senses (meanings) exist: picked—as in selected from among several choices, and picked—as in grasped. Notice that the word following, `UP`, selects the appropriate sense for `PICKED`—grasped. Thus, to remove the ambiguity of the word `PICKED` requires chunks of knowledge on how that word can be used. That knowledge is captured in small procedures called "expectations" or "demons."

Demons are procedures that encapsulate specific elements of domain knowledge to help the parsing process along. Thus, a given demon may be applied to the task of "expecting" or waiting for the word `UP` in order to complete the task of clarifying the word `PICKED`. To deal with words whose sense may be determined from information earlier in the sentence, demons that wait are inappropriate. Demons, in general, are still appropriate, however, since a demon could expect to have the clarifying material already available.

The lexicon with its vocabulary of words allowed in the language being parsed consists of combinations of demon references and needed and simple definitions where ambiguity does not exist. Each word in the sentence above will have a corresponding dictionary entry in the Forth sense; each word will be defined as a colon definition. Each colon definition, then, includes demon and/or definition references, among other information. Parsing then requires typing in the sentence and pressing the return key. Each word in the sentence executes, parsing itself.

Parsing

Parsing a natural language word needs deeper consideration than simple breaking down of a natural language sentence into a new dictionary definition. That would be the case if the result of the parse is the creation of a new executable program. Efforts in this area of "natural programming languages" are a legitimate goal of artificial intelligence research. The parser discussed here is created for a different purpose: intelligent question answering, and scientific discovery support.

The purpose of a parser drives the nature of the memory structures it creates. The parser discussed here creates structures of pointers as a form of "interlingua" from which inferences may be performed, questions answered, and (perhaps) discoveries made. The nature of the pointer structure created by this parser is now discussed.

An Interlingua

A typical parser creates structures comprised of elements which are each different from the original source structure. In contrast, a Forth sentence creates a dictionary entry nearly isomorphic with the original sentence since there is, usually, a one-to-one correspondence between the new dictionary entry and the name of the word compiled, and the cfa's compiled with the desired runtime actions of the words used in the sentence. Isomorphism drops out when numbers, for example, appear. Forth compiles, not the value of the number used in the Forth sentence, but a reference to a primitive runtime word (`LIT`) which pushes the value to the stack when needed.

That brief look inside Forth shows that even a language based on near isomorphism between the source and the parser-created memory structure requires references to two different entities: normal runtime meanings of some words, and references to primitive meanings with others. In fact, it is interesting to note that a full-featured Forth environment can be built entirely out of about a dozen procedural primitives (e.g. `LIT`, `NEXT`, `0BRANCH`, `BRANCH`, `@`, `!` and so on). The rest of a Forth dictionary can be built by creating definitions which either refer to other definitions, or to these primitives, or both. Extending this methodology to a natural language parser only requires that the same types of tools given the Forth environment be provided to the natural language environment. Thus, a set of primitives is required, a dictionary built out of these primitives is required, and something new to Forth: a knowledge base comprised of demons.

These additions exist to build a memory structure that expresses an interlingua, an intermediate "rephrase" of the parsed sentence. It is beyond the scope of this paper to detail an interlingua; a wide range of types of interlingua may be designed according to the needs of the parser. Examples of appropriate interlingua are discussed in [WIL75] and [BRIG84]. The specific version mentioned here for illustration is based on the notion of Conceptual Dependency [SCH77].

Conceptual dependency theory offers a set of transforms on which sentences may be redefined so that later inferences may be facilitated. As an example, the act of speech, or speaking, falls conceptually under the heading of an MTRANS—a memory transfer.

Likewise, conceiving a new idea, or thinking, falls under the MTRANS heading. From that, one would expect that a natural language sentence containing words like "spoke" would cause the parser to generate references to the conceptual primitive MTRANS, just as a Forth reference to a number will cause an internal reference to `LIT`. Notice that "spoke" is ambiguous, and will require demon expectations as discussed below.

Further, just as the Forth primitive `LIT` will follow itself in the new dictionary entry with the value of the number it references, MTRANS must follow itself in the interlingua structure with references to further information. It turns out that `GRASP` happens to be one of the conceptual primitives outlined in conceptual dependency theory. Thus, the sentence:

```
BILL PICKED UP THE HAMMER.
```

may be parsed into a structure that looks something like:

```
GRASP
     ACTOR HUMAN
               NAME BILL
               GENDER MALE
               AGE NIL
     OBJECT PHYS-OBJ
               CLASS TOOL
               NAME HAMMER
     INSTRUMENT MOVE
                    ACTOR HUMAN
                            NAME BILL
                            GENDER MALE
                            AGE NIL
                    OBJECT FINGERS
                    TO PHYS-OBJ
                         CLASS TOOL
                         NAME HAMMER
```

This interlingua representation of the concepts expressed in the sentence provides sufficient information for a program to determine answers to questions like:

Who picked up the hammer?
What was picked up?

The memory structure built tends toward overkill in its representation of the concepts parsed. That's because domain knowledge has been applied to the parsing task such that the act of picking up a hammer is explicitly defined by some of its more general constituent concepts. Picking anything up is an act of grasping; an actor performs the act, and performs it on some object, and uses an instrument—fingers, in this case—in the performance. There's more. It isn't enough for a computer memory structure to retain the notion that fingers do the grasping; fingers are moved to the object being grasped. One could carry that type of reasoning on to even deeper levels of representation (e.g. `FINGERS` are parts of a `HUMAN`), but the illustration makes clear the notion that the interlinguā must explicitly define the concepts as parsed in order that the real meaning inherent in the concepts may be found by an inference program later.

Consider time, for example. `PICKED` can be parsed with the aid of demons or definitions to represent `TIME IN THE PAST`, or `BEFORE NOW`. `TIME` would then become another "slot" in the `GRASP` frame. A frame [MIN75] can be something akin to a tiny one-dimensional array such that each array cell is called a slot, and each slot gets filled with something important to the knowledge being represented. A GRASP frame is built in the interlingua listed above and its slots are named ACTOR, OBJECT, INSTRUMENT. Another slot, then, can be TIME or whatever class of information is necessary to support the concept being represented.

Slots are filled with pointers. Pointers can point to words in the dictionary, or to other frames. In the above example, there are two pointers to a representation of BILL, and one pointer to another concept, MOVE. These pointers are built either directly by the lexicon entries, or by demons.

The lexicon entry for the word BILL would be built out of a Forth colon word:

```
: BILL HUMAN
          NAME MYNAME
          GENDER MALE
          AGE NIL WATCH-FOR-AGE >DEMONS ;
```

This entry into the lexicon illustrates how each definition can look like a frame with slots and entries. Further, the word illustrates entry of a default slot value: NIL, and a demon reference associated with expecting some form of age reference later in the story. A slot filled with a default value is called a gap.

The following shows the structure of a complete parser as a memory map:

```
Low memory: Forth system
                 :
            Parser support words
                 :
            Primitive definitions
                 :
            Demon definitions
                 :
            Lexicon
                 :
Hi memory:  Interlingua structure
```

This memory map illustrates the interlingua data structure built into high memory, but it could just as easily be built into an array allocated somewhere in the Forth dictionary. The ordering of definitions, however, is the key to the structure of this parser; primitive definitions like the concepts MTRANS, GRASP, MOVE, HUMAN, and so on are required first. Demons are next built, followed, finally, by the dictionary of words allowed in the natural language being parsed. From this structure, which mimics the structure of Forth in many ways, the lexicon entries are able to build upon all the structure before. Thus, a definition for the ambiguous word PICKED might look like:

```
: PICKED
      GRASP? ( a demon )
      IF GRASP ( a primitive )
      ELSE SELECT ( a primitive )
      THEN RUNDEMONS ;
```

PICKED ends its definition with a call to a parser support word RUNDEMONS. This brings us to a discussion of demons.

Demons

The word PICKED requires clarification, since it has at least two different meanings, only one of which is suitable in a given place in a given sentence. In this particular word, one needs to wait to see if PICKED is followed by UP. If it is, the primitive definition associated with GRASP is used. Otherwise (as defined above) the primitive SELECT is appropriate. But, waiting for a later resolution is, in fact, the same as a deferred execution, and that requires either multiple passes of the parser, or some form of deferred definition. With the method to be discussed here, deferred definition is used, and the definition of the word PICKED presented above will not work.

The central design concept used with expectations is that the defining word which calls any given expectation demon "spawns" that demon. That is, the cfa of the demon along with some information that demon will need to run properly will be pushed into an execution array (an agenda). The word RUNDEMONS will run all cfa's in that array each time it is called. Some demons will fire by satisfying their needs (finding whatever they "expect') and others will not get what they need. Those demons which fire perform whatever task they must, and then kill themselves. Actually, they return a True flag to the inference engine routine that runs them; they are then removed from the array. Demons which do not find what they are looking or waiting for simply return False and remain in the array for a later attempt at satisfaction.

One demon can serve many masters. The interlingua defined above showed two references to the HUMAN, BILL. Each reference to BILL spawned the demon that waits for an age reference. It's the same demon for both references, but the slots to be filled are different. Thus, the spawning of a demon implies sending along the address of the slot to be filled by that demon, or other such information as a demon might need. Notice that the two references to an AGE demon spawn demons that never fire since there is no age reference in the sentence. Some demons, like those, never actually fire, and the slot remains filled with the default value, NIL. Some demons can die by finding the period at the end of the sentence.

Finally, definitions in the lexicon must be designed to work with the demons they spawn. The definition of PICKED above illustrates the desired performance of that word, but cannot work with the demon GRASP? since that demon must wait for the next word, UP. Several approaches to the solution to this problem may be derived, but one approach is interesting: the process of spawning a demon that must wait includes sending the demon cfa to the execution array, and sending along a gap address called MYGAP and the cfa of the calling word such that the demon itself can execute the word. The word PICKED, then, would have two execution bodies, one which fires when the word is scanned by Forth in a sentence, and one which fires later under command of the demon it spawned. This approach might reduce to specializing the demon such that it fires GRASP or SELECT, and PICKED simply spawns the demon, and calls RUNDEMONS. A definition for PICKED would then read:

```
: PICKED  GRASP? >DEMONS RUNDEMONS ;
```

There are several approaches to demon design in an expectation parser. The demons apply domain knowledge to the task of building a memory structure comprised of pointers to concepts and primitives. We now discuss the primitive definitions.

Primitives

This example parser uses conceptual dependency primitives. The purpose of the primitives is to give the final "meaning" to the interlingua being built by the parser. The Forth analogy holds well here because those dozen procedural Forth primitives mentioned earlier are, in fact, the only routines (in a basic Forth system) which execute and cause the computer behavior designed into the Forth program. So, just like multiplication can be built out of the primitive add routine, the lexicon can be built out of semantic primitives. The definition for the GRASP semantic primitive could look like:

```
: GRASP  MODE @
    IF ." GRASP " CR %SHOW
    ELSE NEWCONCEPT
         ACTOR NIL GAP DUP DEMON1 >DEMONS
         OBJECT NIL GAP DUP DEMON2 >DEMONS
         INSTRUMENT MOVE
                        ACTOR NIL SWAP GAP DEMON3 >DEMONS
                        OBJECT FINGERS
                        TO NIL GAP DEMON3 >DEMONS
         RUNDEMONS
    THEN ;
```

This definition of GRASP affords some discussion of the design of the parser. First, notice that a MODE variable is tested. GRASP has two modes of execution, one when it is working as a parser, and one when the user wants to see the result of the parse. Calling GRASP in the SHOW mode would print the name GRASP, then start the rest of the interlingua printing itself through %SHOW. A third mode might define the behavior of GRASP when logical inferences are being performed on this frame.

Certain words cited in this definition need explanation. NIL defines something called a gap, a space allocation in the interlingua structure with a default pointer to the PRINTtime NIL definition, and the word GAP leaves the previous gap's address on the stack. Note that in the first ACTOR call, GAP is DUPed. This is a Forth method of passing the gap to another citation of the same information. It's the same ACTOR in both citations in GRASP, and the demon DEMON3 takes two values, the earlier gap, and MYGAP (the gap it is to fill) and simply waits for the earlier gap to go non-NIL, at which time it fires, filling MYGAP with whatever was bound to the earlier gap, and kills itself—never to fire again.

DEMON1 takes MYGAP as a parameter and looks around for a HUMAN, in this case BILL, a concept which already exists in the interlingua since BILL came before PICKED. The interlingua address of the concept HUMAN is bound to MYGAP and DEMON1 dies. Almost immediately, one of the DEMON3 clones fires, binding the second ACTOR gap.

DEMON2 takes MYGAP and looks around for a PHYS-OBJ, for which it must wait. DEMON2 stays alive for a couple of passes before it fires.

DEMON3 is spawned again, taking two parameters, MYGAP and an earlier gap (passed by the DUP in an earlier OBJECT) and, again, waits for the earlier gap to go non-NIL. Thus, the same demon can serve different masters, even in the same definition because spawning makes "clones" and killing only erases a cfa reference of a demon, not the original parent code.

A final note about demons as described here: they leave their cfa's on execution so that this value can be pushed into the execution array with >DEMONS.

NEWCONCEPT sets up the array pointers for the interlingua structure to accept a "new concept." BILL was a NEWCONCEPT. GRASP is a NEWCONCEPT. HAMMER will be a NEWCONCEPT. UP will be a NEWCONCEPT to assist the GRASP? demon, but it will be ignored by the rest of the parser since it has no further meaning beyond clarifying PICKED.

Finally, a simple parser would ignore THE, but a rigorous parser would use it to answer the question:

How many hammers did Bill pick up?

Program Listing

Included at the end of this paper is the original source code used to develop and explore these ideas. It is, in essence, a hack; a program written solely to explore ideas. The ultimate intention of these ideas is inclusion of a natural language interface within an inference engine environment such that the primitives, the lexicon, and the expectation demons actually become a part of the total expert system knowledge base.

The code is written in F-83, a Forth-83 dialect, in this case, running on an IBM PC-class personal computer. Its working memory—the space where concepts of the interlingua are developed during parsing— requires only 800 bytes. The program should be capable of fitting on most 64k computers.

There are three files: the parser, the demons, and the lexicon. They are compiled on top of Forth in that order. Sufficient lexicon entries are available to parse the entire sentence:

```
JOHN PICKED UP THE BALL AND DROPPED IT IN THE BOX
```

It is not clear that the demons are sufficiently developed to handle the entire sentence, and `PRETTYPRINT` is not written to look for more than a single "prime" concept. In this sentence, there are two prime concepts: `GRASP` (the ball by John), and `PTRANS` (the ball, by gravity). The code, as is, provides a useful substrate for studying conceptual dependency theory.

Going Beyond

The ability to parse a single, simple sentence does not by any means make a successful parser. Demons are needed to handle entire dissertations, waiting, sometimes for a long time before their gaps are bound and they die. Further, sometimes simple demons are not enough to parse a sentence that requires significant prior knowledge on the part of the listener—the parser. A sentence set like:

```
I AM HUNGRY. WHERE ARE THE KEYS.
```

requires a substrate understanding of a number of social concepts. Other sentences might describe the trip to an eatery, and the event there. Decoding the massive ambiguities this sentence leaves requires more than simple demons. Scripts, MOPs, TAUs, and a host of other parsing tools may be added to a simple parser as described here. These tools are described in [DYE83].

Conclusion

Forth is a useful substrate environment on which to build a natural language parser. When further combined with inference engines [PAR84] and [RED84], the roots of a powerful applied artificial intelligence environment may evolve.

References

[BRI84] Briggs, R. An approach to deeper expert systems. *RIACS*, NASA Ames Research Center. November 1984.

[DYE83] Michael G. Dyer. *In-Depth Understanding*. Cambridge, MA: MIT Press, 1983.

[MIN75] Minsky, M. A framework for representing knowledge. In *The Psychology of Computer Vision*. Ed. P. Winston. McGraw-Hill, 1975. 211- 277.

[PAR84] Park, J. A consequent-reasoning inference engine for microcomputers. *1984 FORML,* Asilomar.

[RED84] Redington, D. Outline of a Forth oriented real-time expert system for sleep staging: a FORTES Polysomnographer. *1984 FORML*, Asilomar.

[SCH77] Schank, F. The primitive ACTs of conceptual dependency. *TINLAP-1:* 39-41.

[WIL75] Wilks, Y. An intelligent analyzer and understander of English. *Comm. ACM* 18.5 (May 1975).

An earlier version of this paper appeared in the 1985 FORML proceedings.

Manuscript received December 1985.

Appendix A

```
Expectation Parser                                   21 April 85

Based on Dyer parser in IN DEPTH UNDERSTANDING by Michael Dyer.
MIT Press.

TO LOAD:
 load PARSER.BLK
 load DEM1.BLK
 load LEX1.BLK
TO PARSE:
 (PARSE) JOHN PICKED UP THE BALL <cr>
TO PRINT:
 PRETTYPRINT


\ Parser load screen

: WALL ;

2 32 THRU

EXIT

This file loads memory structures, support words, and
  CD Acts (Lexicon primitives).

Another file is next loaded (e.g. DEMON1) for demons.
Finally, another file (LEX1) loads the lexicon used to
parse sentences.  Loaded in this order, the lexicon can
use already-compiled demons.

\ Parser opening words

VARIABLE *WM*  800 ALLOT
VARIABLE MODE
FALSE CONSTANT %PARSING
TRUE CONSTANT %PRINTING
VARIABLE #CONS
VARIABLE CONADR
VARIABLE DEMONLIST 514 ALLOT
VARIABLE MYGAP     514 ALLOT
VARIABLE MYVAR     514 ALLOT

EXIT
```

```
*WM* is working memory consisting of
  20 concepts
  40 bytes per concept (=20 cells per concept)
PARSING & PRINTING are mode values needed because each word
  in the lexicon is mode sensitive
#CONS is number of concepts actually used to parse a sentence
CONADR is a moving array address used during parsing
First cell in each concept in *WM* is a count field.
DEMONLIST, MYGAP, & MYVAR comprise a 3-cell execution list
  used by RUNDEMONS
(DEMON) is an execution array that allows forward reference
  for later-defined DEMONS

\ Parser starting words

: (PARSE) ( -- )
  -1 #CONS +! *WM* 800 ERASE ( fresh memory )
  DEMONLIST 2+ 512 ERASE DEMONLIST DEMONLIST !
  MYGAP MYGAP ! MYVAR MYVAR !
  %PARSING MODE ! ;

: NEWCONCEPT ( -- ) 1 #CONS +!
  *WM* ( base adr ) #CONS @ 40 * + 2+ ( skip count cell )
  CONADR ! ;

: >CON ( n -- ) CONADR @ ! 2 CONADR +! ;
HEX
: ?IGNORE ( n -- n tf ) DUP 8000 AND 0= NOT ; DECIMAL

EXIT

\ Memory structure printing
: SHOWCONCEPT ( adr -- )
  BEGIN DUP @ 0= NOT ( done? )
  WHILE DUP @ EXECUTE 2+
  REPEAT DROP ;
: ?NEWCONCEPT ( adr -- tf ) \ true if adr inside *WM*
  *WM* OVER < SWAP *WM* 800 + SWAP > AND ;
: %SHOW ( adr -- adr ) 2+ DUP @ ?NEWCONCEPT
  IF DUP @ SHOWCONCEPT
  ELSE DUP @ EXECUTE
  THEN ;
: PRETTYPRINT  CR
  %PRINTING MODE !  ( now find count field with 00 )
  *WM* BEGIN DUP @ 0=
         IF TRUE ELSE 40 + FALSE THEN ( no bounds checking )
       UNTIL ( -- adr )  2+ SHOWCONCEPT ;
EXIT
```

```
PRETTYPRINT automatically skips any field marked IGNORE. It
  looks for field with 00 count: the root concept developed
  during the parse.

\ Lexicon support words

: GAP  CONADR @ 2- ;

: MYSELF LAST @ NAME> COMPILE (LIT) , ;  IMMEDIATE

: *CONJ* ;  \ used as tag in demons for AND, etc.

: IGNOR ( -- )  \ a primitive demon - sets the IGNORE bit
  #CONS @ 40 * *WM* + ( ^ current concept ) DUP @ [ HEX ]
  8000 AND SWAP ! ; DECIMAL

: BIND ( conadr gap -- )
  1 OVER +! ( inc count field of concept found )
  2+ ( skip count field ) SWAP ! ( bind value to gap ) ;

EXIT

A GAP is a 2-byte space in the concept where a value will
  be bound.  e.g. - the concept:
    PHYS-OBJ GAP
    where GAP will eventually be bound to the concept BALL

\ Demon support

: >DEMONS ( myvar mygap demoncfa -- )
  DEMONLIST @ 2+ ! 2 DEMONLIST +!
  MYGAP @ 2+ ! 2 MYGAP +!
  MYVAR @ 2+ ! 2 MYVAR +! ;

: DEMON@ ( I -- demoncfa )
  2* DEMONLIST 2+ + @ ;

: MYGAP@ ( I -- gap^ )
  2* MYGAP 2+ + @ ;

: MYVAR@ ( I -- var# )
  2* MYVAR 2+ + @ ;

EXIT


3 lists are used by demons:
        1-    the demon cfa
        2-    a variable for the demon - a gap pointer
        3-    another (optional) variable for the demon

\ Demon support
```

```
: KILLDEMON ( I -- )
  2* DUP DUP DEMONLIST 2+ + 0 SWAP !
  MYGAP 2+ + 0 SWAP ! MYVAR 2+ + 0 SWAP ! ;

: RUNDEMONS ( -- )  256 0
  DO I DEMON@
     IF I MYVAR@ I MYGAP@ I DEMON@ EXECUTE ( -- tf )
       IF I KILLDEMON THEN
     THEN
  LOOP ;

EXIT

Each DEMON gets:
   a GAP pointer - which may be 00 if not needed
   a VARIABLE value - which may be:
         another gap pointer
         a search limit
         00 if not needed

Each DEMON returns a truth flag such that:
   TRUE means DONE - ok to kill
   FALSE means not done, leave for try again later.

RUNDEMONS executes all the cfas in the DEMON list.  Most are
00, and are skipped.
KILLDEMON writes 00 over a used DEMON cfa in the DEMON list.


\ Lexicon  - early concepts
: NIL MODE @
  IF ( printing ) ."  NIL "
  ELSE MYSELF >CON
  THEN ;
: ** NIL ;          \ compile default gap

: FEMALE MODE @
  IF ."  FEMALE "
  ELSE MYSELF >CON
  THEN ;

: MALE MODE @
  IF ."  MALE "
  ELSE MYSELF >CON
  THEN ;

EXIT

\ Lexicon

: GENDER MODE @
  IF ."  GENDER " %SHOW
  ELSE MYSELF >CON
  THEN ;
```

```
: NAME MODE @
  IF ."  NAME " %SHOW
  ELSE MYSELF >CON
  THEN ;

: CLASS MODE @
  IF ."  CLASS " %SHOW
  ELSE MYSELF >CON
  THEN ;

EXIT

\ Lexicon primitives

: HUMAN  MODE @
  IF ."  HUMAN " %SHOW CR
  ELSE MYSELF >CON
  THEN ;

: PHYS-OBJ MODE @
  IF ."  PHYS-OBJ " %SHOW CR
  ELSE MYSELF >CON
  THEN ;

EXIT

\ Lexicon  primitives

: TO  MODE @
  IF ."  TO " %SHOW
  ELSE MYSELF >CON
  THEN ;

: FROM MODE @
  IF ."  FROM " %SHOW
  ELSE MYSELF >CON
  THEN ;

EXIT

NEST is used in words that are followed by runtime binding gaps

\ Lexicon  primitives

: OBJECT  MODE @
  IF ."  OBJECT " %SHOW
  ELSE MYSELF >CON
  THEN ;

: ACTOR  MODE @
  IF ."  ACTOR " %SHOW
  ELSE MYSELF >CON
  THEN ;
EXIT
```

```
\ Lexicon primitives - conceptual dependency

: MTRANS  MODE @
  IF ."  MTRANS " %SHOW
  ELSE MYSELF >CON
  THEN ;

: ATRANS MODE @
  IF ."  ATRANS " %SHOW
  ELSE MYSELF >CON
  THEN ;

: PROPEL MODE @
  IF ."  PROPEL " %SHOW
  ELSE MYSELF >CON
  THEN ;
EXIT

Following primitives are from conceptual dependency theory.
These are the primitive "ACTS" which form the root meaning
of a sentence parse.

\ Lexicon primitives

: PTRANS MODE @
  IF ."  PTRANS " %SHOW
  ELSE MYSELF >CON
  THEN ;

: INGEST MODE @
  IF ."  INGEST " %SHOW
  ELSE MYSELF >CON
  THEN ;

: EXPEL MODE @
  IF ."  EXPEL " %SHOW
  ELSE MYSELF >CON
  THEN ;
EXIT

\ Lexicon primitives

: MBUILD MODE @
  IF ."  MBUILD " %SHOW
  ELSE MYSELF >CON
  THEN ;

: MOVE  MODE @
  IF ."  MOVE " %SHOW
  ELSE MYSELF >CON
  THEN ;

: GRASP MODE @
  IF ."  GRASP " %SHOW
  ELSE MYSELF >CON
  THEN ;
EXIT
```

```
\ Lexicon primitives

: SPEAK MODE @
  IF ."  SPEAK " %SHOW
  ELSE MYSELF >CON
  THEN ;

: ATTEND MODE @
  IF ."  ATTEND " %SHOW
  ELSE MYSELF >CON
  THEN ;

: MOBJ MODE @
  IF ." MOBJ " %SHOW
  ELSE MYSELF >CON
  THEN ;
EXIT
\ Lexicon primitives

: POSS MODE @
  IF ."  POSS " %SHOW
  ELSE MYSELF >CON
  THEN ;

: INSTR MODE @
  IF ."  INSTR " %SHOW
  ELSE MYSELF >CON
  THEN ;
```

Appendix B

```
DEMONS 1
This file loads on top of PARSER1 and supports LEX1
to parse the sentence:
  JOHN PICKED UP THE BALL

\ Demons 1

2 12 THRU

\ Demons 1  look back for HUMAN

: EXP1  ( myvar mygap -- tf {T if ok to kill} )
  FALSE ROT 0 ( gap F var 0 )
  DO I 40 * *WM* + DUP 2+ ( skip to head )
    @ ['] HUMAN = ( look for "HUMAN" )
    IF ( gap F adr ) SWAP DROP ( drop F )
      BIND ( bind concept to gap ) TRUE LEAVE
    ELSE DROP ( drop address, try again )
    THEN
  LOOP IF TRUE ELSE DROP FALSE THEN ;

EXIT
```

```
EXPECTATION DEMONS
EXP1 expects a " human before " which means it starts at
  concept Ø and searches up to a limit of the calling
  concept.
  If human is found, the concept base address where it is
  found is stored (bound) into the gap sent by the caller
  (mygap).  Also, the count field of the concept where
  human is found is incremented to show it is not prime:
    some other concept has referenced it.
  If human is found, the demon instance that finds it
  gets killed.
  If not successful, false is returned, and the demon will
  run again, each time RUNDEMONS is called, until it is
  successful, or until the parse is done.

\ Demons 1  wait for PHYS-OBJ

: EXP2  ( myvar mygap -- tf {T if ok to kill} )
  OVER #CONS @ = NOT
  IF SWAP FALSE SWAP ( gap f con ) 1+ #CONS @ 1+ SWAP
    DO I 40 * *WM* + DUP 2+ ( skip to head )
      @ ['] PHYS-OBJ = ( look for "PHYS-OBJ" )
      IF ( gap F adr ) SWAP DROP ( drop F )
        BIND ( bind concept to gap ) TRUE LEAVE
      ELSE DROP ( drop address, try again )
      THEN
    LOOP IF TRUE ELSE DROP FALSE THEN
  ELSE 2DROP FALSE
  THEN ;
EXIT

EXP2 looks for PHYS-OBJ after its caller.  Thus, this demon
  does not look before, it waits till later to fire.

\ Demons 1 copy a gap when it is non-NIL

: EXP3
  OVER @ ['] NIL = NOT
  IF SWAP @ SWAP ! ( bind ) TRUE
  ELSE 2DROP FALSE
  THEN ;

: EXP6  2DROP TRUE ;  \ a hack

\ Demons 1  look back for PHYS-OBJ

: EXP5  ( myvar mygap -- tf {T if ok to kill} )
  FALSE ROT Ø ( gap F var Ø )
  DO I 40 * *WM* + DUP 2+ ( skip to head )
    @ ['] PHYS-OBJ = ( look for "PHYS-OBJ" )
    IF ( gap F adr ) SWAP DROP ( drop F )
      BIND ( bind concept to gap ) TRUE LEAVE
    ELSE DROP ( drop address, try again )
    THEN
  LOOP IF TRUE ELSE DROP FALSE THEN ;
EXIT
```

```
\ Demons 1  disambiguate PICKED

DEFER GRASP?
\ actual code is (GRASP) in LEX1.BLK
```

Appendix C

```
LEXICON 1

Runs the Lexicon developed in:
  IN DEPTH UNDERSTANDING by Michael Dyer (page 407)

to parse the sentence:
  JOHN PICKED UP THE BALL

\ Lexicon 1

2 16 THRU

EXIT

DEM1 the demons file must be loaded before this file.

\ Lexicon 1

: JOHN MODE @
  IF ."  JOHN "
  ELSE NEWCONCEPT
    HUMAN
      NAME MYSELF >CON
      GENDER MALE
    RUNDEMONS
  THEN ;

EXIT

Typical lexicon entry either prints its own name during
 prettyprint, or it builds a concept entry in the CD
 memory space.

\ Lexicon 1

: FINGERS ."  FINGERS " ;

: M1 NEWCONCEPT
  GRASP
    ACTOR ** #CONS @ ( myvar = search limit) GAP ( mygap )
      ['] EXP1 >DEMONS GAP ( pass below H )
    OBJECT ** #CONS @ ( start search ) GAP
      ['] EXP2 >DEMONS GAP SWAP ( pass X )
    INSTR MOVE ACTOR ** ( H) GAP ['] EXP3 >DEMONS
               OBJECT ['] FINGERS >CON
               TO ** ( X ) GAP ['] EXP3 >DEMONS ;
```

```
EXIT

FINGERS can be expanded to:
  PART-OF  HUMAN  etc.

\ Lexicon 1

: M2 NEWCONCEPT
  MBUILD
    ACTOR ** #CONS @ GAP ['] EXP1 >DEMONS
    MOBJ POSS
           ACTOR ** #CONS @ GAP ['] EXP1 >DEMONS
           OBJECT ** #CONS @ GAP ['] EXP4 >DEMONS  ;

2 CASE: (PICKED) M1 M2 ;

EXIT

A case statement is used since PICKED has two possible
meanings in this lexicon.

\ Lexicon 1

: PICKED
  Ø #CONS @  ( starting concept )
  ['] GRASP? >DEMONS ;

: UP NEWCONCEPT MYSELF >CON IGNOR RUNDEMONS ;

: (GRASP?) ( Ø start concept -- true ) SWAP DROP 1+
  40 * *WM* + 2+ ( next concept head ^ UP ? ) DUP @
  ['] UP =
  IF 1 SWAP 2- +! Ø ELSE DROP 1 THEN
  ( case # ) (PICKED) TRUE ;

' (GRASP?) IS GRASP?
\ grasp was deferred in DEM1.BLK
EXIT

(GRASP?) is the runtime code for GRASP?, a deferred demon.
  If next concept is UP, then PICKED --> GRASP, else MBUILD

PICKED gets set as a concept **after** UP has been set as
  a concept in *WM*.  John is CONO, UP is CON1, GRASP will be
  CON2.

\ Lexicon 1

: THE ;  \ do nothing with this word

: GAME-OBJ ." GAME-OBJ " ;
: BALL MODE @
  IF ."  BALL "
  ELSE NEWCONCEPT
    PHYS-OBJ
      CLASS ['] GAME-OBJ >CON
      NAME MYSELF >CON
    RUNDEMONS
  THEN ;

EXIT

\ Lexicon 1

: AND NEWCONCEPT ['] *CONJ* >CON  IGNOR ;

: GRAVITY ."  GRAVITY " ;

: DROPPED NEWCONCEPT
  PTRANS
    ACTOR ** #CONS @ GAP ['] EXP1 >DEMONS
    OBJECT ** #CONS @ GAP ['] EXP5 >DEMONS GAP ( THG )
    TO ** #CONS @ GAP ['] EXP6 >DEMONS
    INSTR PROPEL
            ACTOR ['] GRAVITY >CON
            OBJECT ** GAP ['] EXP3 >DEMONS ( use THG )
  RUNDEMONS ;

EXIT

The total sentence this lexicon supports is:
  JOHN PICKED UP THE BALL AND DROPPED IT IN THE BOX
The total sentence parsed as an experimental exercise is:
  JOHN PICKED UP THE BALL

\ Lexicon 1

: IT ;   \ do nothing for now

: IN ;   \ do nothing for now

EXIT

\ Lexicon 1

: CONTAINER ."  CONTAINER " ;

: BOX  MODE @
  IF ."  BOX "
  ELSE NEWCONCEPT
    PHYS-OBJ
      CLASS ['] CONTAINER >CON
      NAME MYSELF >CON
  THEN ;

EXIT
```