UPMAIL Technical Report No. 14C

A Grammar Kit In Prolog

Kenneth M. Kahn
revised by Mats Carlsson

February 2 1985

UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science, Uppsala University
P.O. Box 2059
S-750 02 Uppsala Sweden
Domestic 018 155400 ext. 1860
International +46 18 111925
Telex 76024 univups s

The research reported herein was supported by the National Swedish Board for Technical Development (STU) and Atari Research Cambridge, Massachusetts.

Keywords: Prolog, Logic Programming, Natural Language, Education, Artificial Intelligence, Dynamic Graphics

Table of Contents

1. Computers and Children 1
2. Prolog and Language 2
3. An Imaginary Scenario 3
4. Children and Natural Language 7
5. Implementation Issues 7
6. References 9
7. Appendix — Source code of the Grammar Kit 10

Abstract

Prolog was originally developed as a programming language for writing natural language parsers. This paper presents a prototype of a "grammar kit" written in Prolog. The kit consists of tools and examples intended to be used by children interested in exploring language. A grammar written in Prolog using this tool kit can be used not only to parse, but also to generate, sentences. A serious shortcoming of Prolog for children is that when a program misbehaves one must understand its complex operational semantics. A major tool in the grammar kit to alleviate this problem is a dynamic graphical tracer. This allows the user to see a parse tree as it is being built. This tool, in turn, depends heavily on a mechanism for delaying goals built into LM-Prolog. A rather significant side-effect of using the graphical tracer is a better understanding of Prolog's chronological backtracking.

1. Computers and Children

The major goal of this research is to provide children with a rich computational environment for doing interesting natural language projects. While this environment is built upon Prolog, the methodology and philosophy is closer to that of SmallTalk (Goldberg and Ross, 1981) and Logo (Papert, 1980). In other words, children are encouraged to do relatively long-range projects in a very rich and powerful environment, and in the process, learn about the domain of the project (languages, translation, and grammar in this case), computational concepts such as variables, recursion, representation, process, etc. and learning itself.

This research, in contrast with other projects dealing with Prolog and children (Ennals, 1982), is based upon the belief that the children should understand how Prolog works in order to use it. The pragmatic reason for this so that the children can cope with programs that don't behave properly. To some extent this problem is a consequence of the fact that Prolog does not live up to the ideal of logic programming very well. The order of clauses and goals matters. Shortcomings of the language are filled by extra-logical primitives. It may be the case that the most important thing that a child can get from using a computer is an understanding of process. A more powerful and cleaner logic programming language which comes closer to declarative programming than Prolog may actually be inferior from this point of view.

There are at least three ways of helping children to understand the procedural interpretation of Prolog programs.

- Develop a clear and simple operational model of Prolog and a methodology for introducing children to it.
- Translate "troublesome" Prolog programs into a notation which the children can already understand procedurally. For example, the program could be automatically translated into Logo and studied in that form.

- Provide tools for observing Prolog in action. This is the approach presented in this paper. The major tool is a dynamic backtracking graphical trace facility.

2. Prolog and Language

For doing projects such as building parsers, generators, translators, question-answering systems and the like Prolog has the following advantages.

- A parser can be used as a generator or visa versa. A translator works in either direction.
- A program can be run on partial inputs. A sentence can be parsed with a few words unknown or generated where it is constrained to be a certain length or contain certain words.
- The grammar can be described relatively declaratively without using a special parser.
- An "ask about" facility which asks the user if something cannot be found in the database and then adds the answer to the database is important for being able to use a system before it is complete. The sample parser presented in this paper, for example, has no vocabulary at all, it just knows enough to ask.
- The Prolog database provides a natural and uniform way of connecting a natural language system and a database.
- Pattern matching makes it easier to deal with structures such as lists and parse trees.

On the negative side the implicit control does not always do the right thing. An expert Prolog programmer knows how to anticipate or detect such situations and to transform the program to fix the problem. A child cannot be expected to do that, at least without some simple yet powerful tools. An always up-to-date graphical image of the current state of computation is one such tool.

There has been much work within the Prolog community on devising grammars that translate straight-forwardly into Prolog programs (Pereira and Warren, 1980). The grammar kit contains a predicate called → which transforms its arguments into an ordinary Prolog clause and adds it to the database. Consider the LM-Prolog (Carlsson and Kahn, 1983) definitions of the rule that a verb phrase can consist of a verb followed by a noun phrase, first with, then without, →. (An LM-Prolog statement is a non-atomic s-expression where variables are distinguished by beginning with a ?).

```
(→ (verb-phrase (vp ?verb ?np))
   (verb ?verb)
   (noun-phrase ?np))
```

In DEC-10 Prolog, the above rule would, modulo Prolog syntax and placement of arguments, translate into a clause extending the verb-phrase predicate which is a relation between a tree structure and a part of a list of words:

```
(assert ((verb-phrase ?words0 ?words (vp ?v ?np))
         (verb ?words0 ?words1 ?v)
         (noun-phrase ?words1 ?words ?np)))
```

Our grammar predicates are actually relations between tree structures, parts of a list of words, and parts of lists of graphics commands. Our → predicate translates the above rule into the following:

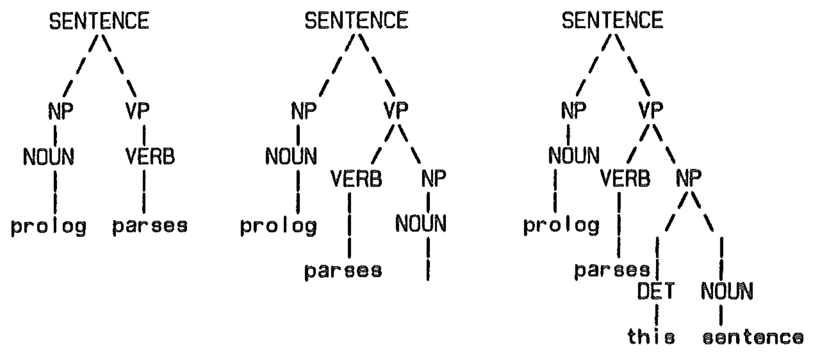
```
(assert
 ((verb-phrase ?words0 ?words
               ((:here ?x ?y ?) (:place-turtle ?x ?y 240.0) . ?coms0) ?coms
               (vp ?v ?np))
  (display-constituent ?coms0 ?coms1 verb)
  (verb ?words0 ?words1
        ?coms1 ((:place-turtle ?x ?y 120.0) . ?coms2)
        ?v)
  (display-constituent ?coms2 ?coms3 verb)
  (noun-phrase ?words1 ?words ?coms3 ?coms ?np)))
```

The predicate display-constituent is responsible for generating graphics commands for displaying a constituent. It is discussed in Chapter 5.

3. An Imaginary Scenario

Consider the following scenario of a child working on a simple project using the grammar kit. The child begins by loading the small sample grammar which is presented below. He or she begins by making up a simple sentence and having the machine parse it. The student enters the LM-Prolog goal `(parse (Prolog parses this sentence))`. The system then asks a series of questions such as `Is (WORD PROLOG NOUN) true?` and `Is (WORD PARSES VERB) true?` Both positive and negative answers are stored so that a question is only asked once. The ability to start with an empty dictionary and build it up interactively as needed is very useful in this application. The child can begin to use his or her grammar immediately without putting a large effort into typing in a dictionary. The alternative of providing a ready-made dictionary to the children strongly constrains the sort of grammars they can write.

As questions are being asked a parse tree is being drawn on the screen as depicted below. It is important to note that the tree is grown as Prolog searches for parses and when it backtracks a part of the tree disappears.



Three Snapshots of the Display

At this point the child is encouraged to try a few more sentences, such as `(the big girl loves the silly boy)` and `(star trek lives)`. Then the system is asked to generate sentences by entering `(parse ?s)`. It generates `(prolog parses)`, `(sentence parses)`, `(girl parses)`, `(boy parses)`, `(trek parses)` and then asks the user `Is there another ?WORD such that (WORD ?WORD NOUN) is true?`. If the child answers no, then more sentences such as `(prolog loves)`, `(sentence loves)` and `(sentence parses prolog)` are generated. These are rather dull sentences, but the child can partially specify the desired sentences. For example, `(and (length 7 ?s) (member prolog ?s) (member silly ?s) (parse ?s))` will generate sentences seven words long containing the words prolog and silly.

Looking at the generated sentences one notices that some of them are not quite right. For example, `(Prolog lives silly star star star trek)` is generated. The parse tree reveals that `lives ... trek` is considered a verb phrase. At this point the sample grammar might be presented to the child and its rules explained if they haven't been already. The child can then perhaps be lead to see a point in distinguishing between transitive and intransitive verbs and can go back and edit the grammar accordingly.

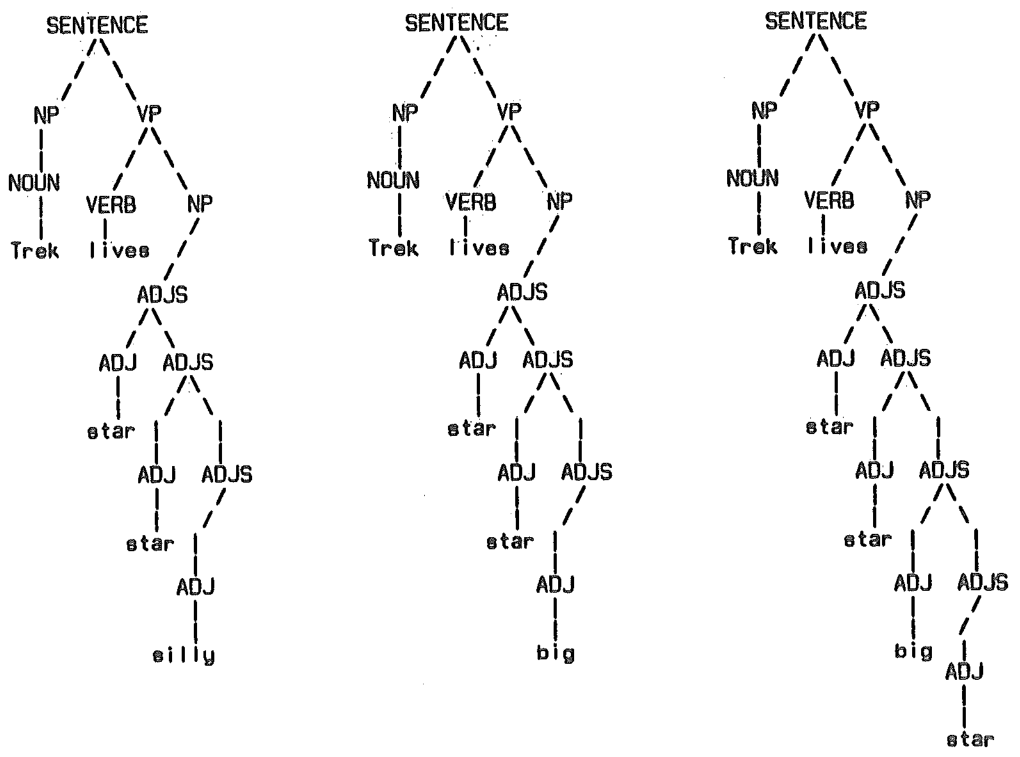
The entire sample grammar follows:

```
(→ (sentence (sentence ?np ?vp))
   (noun-phrase ?np)
   (verb-phrase ?vp))


(→ (noun-phrase (np ?noun)) (noun ?noun))
(→ (noun-phrase (np ?det ?noun))
   (determiner ?det)
   (noun ?noun))
(→ (noun-phrase (np (adjectives . ?adjs) ?noun))
   (adjectives ?adjs)
   (noun ?noun))
(→ (noun-phrase (np ?det (adjectives . ?adjs) ?noun))
   (determiner ?det)
   (adjectives ?adjs)
   (noun ?noun))


(→ (adjectives (?adj)) (adjective ?adj))
(→ (adjectives (?adj . ?more-adjs))
   (adjective ?adj)
   (adjectives ?more-adjs))


(→ (adjective (adjective ?word))
   (is-word ?word adjective))


(→ (determiner (determiner ?word))
   (is-word ?word determiner))


(→ (noun (noun ?word))
   (is-word ?word noun))


(→ (verb-phrase (verb-phrase ?verb))
   (verb ?verb))
(→ (verb-phrase (verb-phrase ?verb ?np))
   (verb ?verb)
   (noun-phrase ?np))


(→ (verb (verb ?word))
   (is-word ?word verb))
```

While generating sentences, one notices that we often get adjectives repeated such as `(the silly silly silly silly Prolog parses big sentences)`. An interesting exercise is how to prevent this. One can place a restriction on a rule by "calling" Prolog directly. We can restrict the rule for adjectives as follows:

```
(→ (adjectives (?adj . ?more-adjs))
   (adjective ?adj)
   (adjectives ?more-adjs)
   (call (not (member ?adj ?more-adjs))))
```

This not only prevents the system from generating sentences with the same adjectives occurring twice in the same noun phrase but prevents the system from generating any sentence with more than two adjectives in a phrase. This is far from obvious by inspecting the program. Running it causes it to loop after generating sentences with just one adjective. What is wrong is apparent as one watches the tree being grown as it loops. A few snapshots of the screen illustrate this.



It keeps looking for longer and longer lists of adjectives in the hope that some will pass the constraint (not (member ?adj ?more-adjs)). The problem is that at this point the first two adjectives are the same so adding more adjectives will never help. The solution to this problem is left as an exercise to the reader.

Except for this attempt to prevent duplicate adjectives, we have not made use of the argument that was in the sample grammar for building a list structure corresponding to the parse tree. The graphics facility does not use it, so one may wonder why one needs the parse tree since it is displayed on the screen. It is useful for translation, searching for ambiguous sentences, and building a natural language interface. A parser must build structures appropriate to its use and the sample grammar provides an example of how to do this.

4. Children and Natural Language

One may wonder how a child could be expected to build natural language interfaces when researchers in the field find it difficult. A simple natural language interface to a simple program can be very exciting to build and use. I once worked with a not especially gifted 13 year-old using a grammar kit I had written in Logo to build a system that could accept and execute English commands like "Draw me a big house" or "Put a long house in the middle" (Kahn, 1975 and 1977). I am confident that the system we constructed would have been simpler, more powerful, and easier to build if it used the Prolog grammar kit described here.

In the interests of simplicity, the sample grammar presented here is a context-free grammar. The basic mechanism with its arguments to the rules allows one to build up and use during parsing arbitrarily complex structures. It is important that children do not bump into the limitations of a system too quickly. Older or clever children might want to pass beyond simple grammars and deal with more sophisticated grammatical concepts (such as number or tense) or move into semantics. Or very special purpose grammars may be most appropriate. For example, in Kahn (1975) a grammar for greetings was built by a 13 year-old. Definite clause grammars are well-suited for these kinds of experimentation.

5. Implementation Issues

The grammar kit is very small since it builds upon some generally useful LM-Prolog utilities, especially the "freeze", "ask about", and "backtracking turtle graphics" packages. The code is presented in the Appendix.

The implementation of the tree-drawing trace facility was not simple. The graphics had to work without touching the user's code. The tree had to be drawn (and undrawn) as the processing was being performed. A much simpler, but much less useful, system would just draw the parse tree after a successful parse. The tree-drawing facility exploits LM-Prolog's multiple databases. Predicates such as display-constituent exist in one version with graphics and one dummy version without.

A problem in drawing the tree is to keep it readable. Without special precautions, its very easy to get labels written on top of each other, to get arcs crossing, and to go off the edge of the display. An earlier implementation of the grammar kit was based upon the idea that the tree drawing should be non-deterministic and that these sorts of æsthetic problems should cause failures. This often produced good-looking trees but lead to insurmountable problems of separating the non-determinism of the tree-drawing from the non-determinism of the application. If one wanted another parse of a sentence, one saw first a large number of alternative ways of displaying the current parse before another one would be generated. The current implementation checks only for label collisions. When one is detected it simply makes the arc involved longer and tries again. A good topic for further research is to find a more elegant solution to this problem. Another is to generalize the tree-drawing facility to support other depictions of the current state of the computation. A maze searching program could draw the current path through the maze, an arbitrary Prolog program could be depicted as an "and-or" tree, etc..

The practical difficulty with the grammar kit presented here is that it requires a powerful computer environment. The kit is implemented in LM-Prolog which runs only on Lisp Machines (Moon *et al.* 1983). A fast machine is needed for the dynamic graphics. A large address space is needed for the entire system. Consequently this kit has not been tested with children. It is only a matter of time, however, before personal machines of sufficient power are widely available to children.

This paper has attempted to convey three ideas about natural language processing and children.

- That much of the work being done on natural language processing within the Prolog community is both very powerful and well-suited for children.
- That dynamic graphics is very valuable as a tool for observing and debugging complex processes.
- That one should provide kits for children, so that they can do exciting large-scale projects without having to start from scratch.

6. References

Carlsson M., Kahn K.M., "LM-Prolog User Manual", UPMAIL, Uppsala University, Technical Report No. 24, October 1983.

Ennals, R., "Teaching logic as a computer language in schools", Proceedings of the First International Logic Programming Conference, Marseille, France, September 1982.

Goldberg, A. and Ross, J., "Is the Smalltalk-80 System for Children?", *Byte* Vol. 6, No. 8, August 1981.

Kahn, K., "A LOGO Natural Language System", LOGO Working Paper 46, MIT AI Laboratory, December 1975.

Kahn, K., "Three Interactions between AI and Education" in *Machine Intelligence 8, Machine Representations of Knowledge*, eds. Elcock E. and Michie, D., Ellis Horwood Ltd. and John Wylie & Sons, 1977.

Papert, S.,*Mindstorms — Children, Computers, and Powerful Ideas*, Basic Books, Inc., New York, 1980.

Pereira, F. and Warren, D., "Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks", *Artificial Intelligence*, Vol 13. No. 2, May 1980.

Moon D., Stallman R.M., Weinreb D., "Lisp Machine Manual", MIT AI Laboratory, Cambridge MA, 1983.

7. Appendix — Source code of the Grammar Kit

The syntax of LM-Prolog differs from other Prolog implementations. Variables are symbols beginning with ?. Terms are Lisp s-expressions. Define-predicate is used to group together different clauses for the same predicate. Comments are preceded by ;.

Given a grammar rule, → adds four extra arguments, as discussed in Chapter 1, and asserts it as an ordinary Prolog clause. The extra arguments are added to the head of the clause and to each statement in the body unless it is an explicit call to Prolog.

```
;;; -*- Mode: Lisp; Package: Puser; Base: 10.; Options: ((World Grammar)) ; -*-
;;; (C) Copyright 1983,1984,1985, Uppsala University
;;; Cleaned up using the Turtle stream by Mats Carlsson.
;;; Compiler.


(define-predicate define-rules
  (:options (:lisp-macro-name define-rules))
  ((define-rules ?name . ?rules)
   (define-predicate ?name :(options (place-clauses after) (type dynamic)))
   (and . ?rules)))


(define-predicate →
  (:options (:lisp-macro-name →))
  ((→ (?part-of-speech . ?arguments) . ?constitutents)
   (translate→ ?new-constitutents ?constitutents ?S0 ?S ?P0 ?P)
   (assert ((?part-of-speech ?S0 ?S ?P0 ?P . ?arguments) . ?new-constitutents))))


(define-predicate translate→
  ((translate→ ?new-constituents ?constituents ?s0 ?s ((:here ?x ?y ?) . ?p0) ?p)
   (length→ ?l 0 ?constituents)
   (trans-clause→ ?new-constituents ?constituents ?s0 ?s ?p0 ?p 0 ?l ?x ?y)))


;;Count no. of constituents, don't count explicit Prolog calls.
(define-predicate length→
  ((length→ ?l ?l ()))
  ((length→ ?l ?l0 ((call . ?) . ?rest))
   (length→ ?l ?l0 ?rest))
  ((length→ ?l ?l0 (? . ?rest))
   (sum ?l1 ?l0 1)
   (length→ ?l ?l1 ?rest)))
```

```
(define-predicate trans-clause→
  ((trans-clause→ () () ?s ?s ?p ?p ?n ?n ? ?))
  ((trans-clause→ ;;Translate a general predicate call
                  (?call . ?new-rest)
                  ((CALL ?call) . ?rest)
                  ?s0 ?s ?p0 ?p ?n0 ?n ?x ?y)
   (trans-clause→ ?new-rest ?rest ?s0 ?s ?p0 ?p ?n0 ?n ?x ?y))
  ((trans-clause→ ;;Translate a word class occurrence
                  ((word ?word . ?args)
                   (display-terminal ?p0 ?p1 ?word) . ?new-rest)
                  ((IS-WORD ?word . ?args) . ?rest)
                  (?word . ?s0) ?s
                  ((:placeturtle ?x ?y ?theta) . ?p0) ?p
                  ?n0 ?n ?x ?y)
   (angle→ ?n ?n0 ?n1 ?theta)
   (trans-clause→ ?new-rest ?rest ?s0 ?s ?p1 ?p ?n1 ?n ?x ?y))
  ((trans-clause→ ;;Translate a terminal occurrence
                  ((display-terminal ?p0 ?p1 ?word) . ?new-rest)
                  ((TERMINAL ?word . ?) . ?rest)
                  (?word . ?s0) ?s
                  ((:placeturtle ?x ?y ?theta) . ?p0) ?p
                  ?n0 ?n ?x ?y)
   (angle→ ?n ?n0 ?n1 ?theta)
   (trans-clause→ ?new-rest ?rest ?s0 ?s ?p1 ?p ?n1 ?n ?x ?y))
  ((trans-clause→ ;;Translate a non-terminal occurrence
                  ((display-constituent ?p0 ?p1 ?rule)
                   (?rule ?s0 ?s1 ?p1 ?p2 . ?args) . ?new-rest)
                  ((?RULE . ?args) . ?rest)
                  ?s0 ?s
                  ((:placeturtle ?x ?y ?theta) . ?p0) ?p
                  ?n0 ?n ?x ?y)
   (angle→ ?n ?n0 ?n1 ?theta)
   (trans-clause→ ?new-rest ?rest ?s1 ?s ?p2 ?p ?n1 ?n ?x ?y)))

(define-predicate angle→
  ((angle→ ?n ?n0 ?n1 ?theta)
   (sum ?n1 ?n0 1)
   (lisp-value ?theta (compute-theta '?n0 '?n) :dont-invoke)))

(defun compute-theta (n out-of)
  (cond ((> out-of 1) (*120. (- 2 (// (float n) (float (1- out-of))))))
        (t 180.)))
```

The predicate word is by default defined to query the user.

```
(define-predicate is-word
  ((is-word (?word . ?left) ?left ?word . ?arguments)
   (word ?word . ?arguments)))

(ask-about word) ;;this is the definition of the dictionary!!
```

The addition of extra arguments can be confusing to a novice. Using the predicate parse the extra arguments are taken care of automatically.

```
(define-predicate parse
  ((parse ?sentence)
   (parse ?sentence ?))
  ((parse ?sentence ?tree)
   (parse ?sentence ?tree ?))
  ((parse ?sentence ?tree ?graph0)
   (turtle-stream ?graph0)
   (= ?graph0 ((:hideturtle) (:placeturtle 0.0 500.0 180.0) . ?graph1))
   (display-constituent ?graph1 ?graph sentence)
   (sentence ?sentence () ?graph () ?tree)))
```

The following is the tree-drawing facility of the grammar kit. Graphics is a way of turning on and off the tree drawing facility by changing the list of current worlds (databases). One can also "bind" the change by saying `(without-world :graphics (parse ...))` or `(with-world :graphics (parse ...))`.

```
(define-predicate graphics
  ((graphics on) (add-world :graphics))
  ((graphics off) (remove-world :graphics)))
```

The graphics is built upon a mechanism for delaying goals and a backtracking turtle. A backtracking turtle is like a Logo turtle (Papert, 1980) but any change to its state is undone upon backtracking. The grammar continuously instantiates a list of graphics commands for the turtle, which executes them as they are instantiated. This is handled by the predicate turtle-stream. Should the program backtrack, commands that are not instantiated any more are undone. This provides an ideal mechanism for drawing parse trees. The tree is drawn from the top of the screen downward.

To carefully display some text, we see first if the turtle is near some text already placed. If it is, the turtle goes forward and we try again. Otherwise, the text is displayed and the current turtle position is added to the database. Assume is an LM-Prolog predicate which adds its argument to the database and removes it upon backtracking or if processing is aborted.

```
(define-predicate display-constituent
  ((display-constituent ((:forward 40) . ?p0) ?p1 ?name)
   (display-terminal ?p0 ((:forward 40) . ?p1) ?name)))

(define-predicate display-terminal
  ;;Dummy version without graphics
  ((display-terminal ?p ?p ?)))

(define-predicate display-terminal
  :(options (world graphics))
  ((display-terminal ((:here ?x ?y ?) . ?p0) ?p ?name)
   (cases ((near-some-text ?name ?x ?y)
           (= ?p0 ((:forward 40) . ?p1))
           (display-terminal ?p1 ?p ?name))
          ((= ?p0 ((:markv ?name) . ?p))
           (assume ((text-displayed-at (?x ?y) ?name)) :graphics)))))
```

```
(define-predicate text-displayed-at) ;;maintained by CAREFUL-DISPLAY above

(define-predicate near-some-text
  ((near-some-text ?text ?x ?y)
   (size-of-text (?height ?length-of-text) ?text);;N.B. TV units.
   (text-displayed-at (?others-x ?others-y . ?) ?other-text)
   (lisp-value ?y-distance (*0.34 (abs (- '?y '?others-y))) :dont-invoke)
   (< ?y-distance ?height)
   (lisp-value ?x-distance (*0.68 (abs (- '?x '?others-x))) :dont-invoke)
   (size-of-text (? ?length-of-other-text) ?other-text)
   (sum ?2width ?length-of-text ?length-of-other-text)
   (< ?x-distance ?2width)))
```