MASSACHUSEITS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 213 December 1980

A STEP TOWARDS AUTOMATIC DOCUMENTATION

Claude Frank*

Abstract: This paper describes a system which automatically generates program documentation. Starting with a *plan* generated by analyzing the program, the system computes several kinds of summary information about the program. The most notable are: a summary of the cliched computations performed by the loops in the program, and a summary of the types and uses of the arguments to the program. Based on this information, a few English sentences are produced describing each function analysed.

* Visiting Scientist on leave from Schlumberger-Doll Research.

CONTENTS

FIGURES

1. Introduction

Automatic documentation is desirable for many reasons. One of them is that many existing programs are poorly documented if at all and yet need to be maintained. Some time after the author wrote the code, he (or she) may already have difficulty modifying it himself (or herself), and let us not speak about somebody else having to do it, as is often the case. In the best situation, where ample comments are present in the code, there is unfortunately no guarantee, and no way to check, that they correspond totally to the actual code. The same is also true for documentation separate from the code. Experience has taught many people that it is usually even less up-to-date and therefore less reliable than in-line comments.

One very attractive solution to this problem is to have the maintenance done with the help of an interactive system which could, among other things, answer the questions of the programmer in charge of the maintenance. In order to generate these answers, the system would use the code as input and generate simple English sentences to guide the maintainer.

2. Present work and results

The work reported here is a step towards solving this problem. A set of programs analyze a piece of code and generate for it a few English sentences which describe what it does. It is integrated into a much larger effort by Rich and Waters called the Programmer's Apprentice or PA [1,2,4]. The system described here uses the plan analyzer system developed by Waters [5,6] in order to create a plan corresponding to the program to be documented. A plan is an explicit representation of all of the control flow and data flow which exists in a program and is a programming language independent representation. This plan is then further analysed in order to create documentation. The plan analyzer can construct plans for programs written in LISP, FORTRAN, and COBOL.

At the outset of this work, one major problem was to find out what it is that one wants to say about programs. I solved the problem by taking some code and writing a few English sentences which I thought were helpful and could be generated based on the information contained in the available plans. Of course, I had to compromise and stay within the level of abstraction offered in the plans.

I will now present an example set of MacLisp programs along with the automatic documentation produced. The items which are capitalized in the text correspond to type information or pieces of code: this implements a partial mapping between the code and the generated documentation meant to help the

programmer.

```
(define hash (symbol)
  (arrayfetch tbl (abs (remainder (maknum symbol) tblsize))))

Function HASH has 1 input-argument , 2 free-variable-inputs.
The input argument is SYMBOL used in MAKNUM.
The input free variables are:
- VECTOR TBL used in ARRAYFETCH
- NUMBER TBLSIZE used in REMAINDER.
The return value is part of TBL and is generated in ARRAYFETCH.
It is an expression.
```

```
(define lookup (symbol)
   (prog (bkt)
         (setq bkt (cdr (hash symbol)))
      lp (or bkt (return nil))
         (cond ((eq (car bkt) symbol)
                (return t)))
         (setq bkt (cdr bkt))
         (go lp)))

Function LOOKUP has 1 input-argument , 2 free-variable-inputs.
The input argument is SYMBOL used in HASH and/or EQ.
The input free variables are:
- TBL used in HASH
- TBLSIZE used in HASH.
The return value is the constants NIL or T.
It is a loop with 2 exits and the following 2 components:
- traversal of the list BKT
- an additional end of loop test.
```

```
(define insert (symbol)
   (prog (bucket)
         (setq bucket (hash symbol))
         (rplacd bucket (cons symbol (cdr bucket)))))

Function INSERT has 1 input-argument , 2 free-variable-inputs and has side-effects.
The input argument is SYMBOL used in CONS and/or HASH.
The input free variables are:
- TBL used in HASH and/or RPLACD
- TBLSIZE used in HASH.
The return value is the constant NIL.
Variable TBL is side-effected in RPLACD.
It is an expression.
```

```
(define delete (symbol)
   (prog (bkt obkt)
         (setq bkt (hash symbol)
               obkt bkt)
      lp (setq bkt (cdr bkt))
         (or bkt (return nil))
         (cond ((eq (car bkt) symbol)
                (rplacd obkt (cdr bkt)))
               (t (setq obkt bkt) (go lp)))))
```

```
Function DELETE has 1 input-argument , 2 free-variable-inputs and has side-effects.
The input argument is SYMBOL used in EQ and/or HASH.
The input free variables are:
- TBL used in RPLACD and/or HASH
- TBLSIZE used in HASH.
The return value is the constant NIL.
Variable TBL is side-effected in RPLACD.
It is a conditional with 2 branches:
- the predicate is a loop with 2 exits and the following 2 components:
     - an additional end of loop test
     - traversal of the lists OBKT and BKT
- the 1st branch is an expression.
```

The automatic documentation generated here is far from being perfect but, with the help of the mnemonics, one gets a good idea of what the code does.[1] Of course, it would be infinitely nicer to be able to say *this is a hash table.* Yet, the detailed information given here is useful when modifying the code and this points to a more general problem: a system should be able to show several levels of detail so that the programmer can look at the code from different points of view, depending on his or her needs.

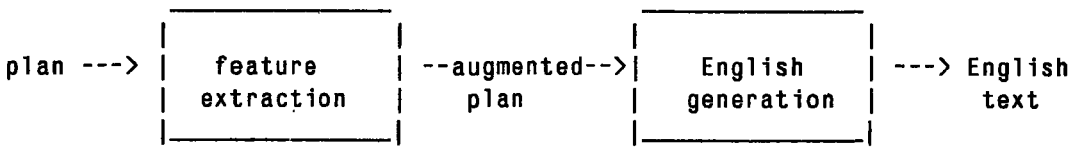
3. Overview of the algorithm

In order to isolate the English generation part, the program is partitioned into two phases:

- a feature extraction program
- an English sentence generator.

1. Here is some idea about the performances on a DEC-10 with a KA-10 processor: 50 seconds of CPU time (phase 1 is 37 seconds and phase 2 is 13 seconds). These measurements were made with the plans already loaded in core.

One important reason for doing this, is that a relatively simple English generation mechanism [3] is used, which could ultimately be replaced by a more sophisticated one. Also for practical reasons, this setup is much easier to handle. The two phases communicate by means of plans to which assertions have been added to reflect the feature extraction.



The data used as an example is a group of functions which implement data storage and retrieval for a simple in-core data base written in MacLisp. The code is shown in section 2. Feature extraction was also tested on several other examples: vector and matrix traversing, and a much more complicated data base package.

4. Feature extraction

The main reason for this phase is that, although a plan contains the complete information about a program, it is so detailed that it cannot be readily used for documentation purposes. The area where this is most flagrant is loops. Therefore, it was decided to try and come up with features of interest for a programmer in order to characterize loops.

The key idea is to recognize cliches at a level which exists in plans as described in Waters [5,6] and no effort is made to recognize higher level abstractions. Furthermore the terminology and conventions for plans defined in [5] are used here.

4.1 Recognized loop features

The features which are actually recognized are:

- number of exits of loops
- nesting of loops represented by a tree structure
- loop cliches such as a counting loop or a list enumerating loop.

4.2 An example

In actual loops, there are combinations of many of these features. For example, in a loop which counts the number of elements of a list such as:

```
(do ((l list (cdr l))
     (n 0 (1+ n)))
    ((null l) n))
```

counting and list enumeration attributes are recognized. Furthermore, it has one exit and a nesting level of one.

4.3 Counting loop

Counting is the attribute of a loop which generates an arithmetic series of numeric values until a certain maximum (or minimum) value is reached. It must have:

- a generator which is initialized with a numeric value (e.g. constant) called *first*, and whose operator is addition or substraction of another numeric constant value called *increment*,
- a terminator which tests against a third numeric value called *last*.

The generator is also recognized if it uses the Lisp operators for incrementing by one (e.g. 1+, 1-). This loop is then qualified with the assertion:

```
(loop-count first increment last).
```

If only the first property holds, then the attribute is called count generator and the assertion is of the form:

```
(gen-count first increment NIL).
```

Figure 1 shows a partial and simplified plan for the example of 4.2. Only the required data-flow links and operators needed to recognize a count generator are represented.

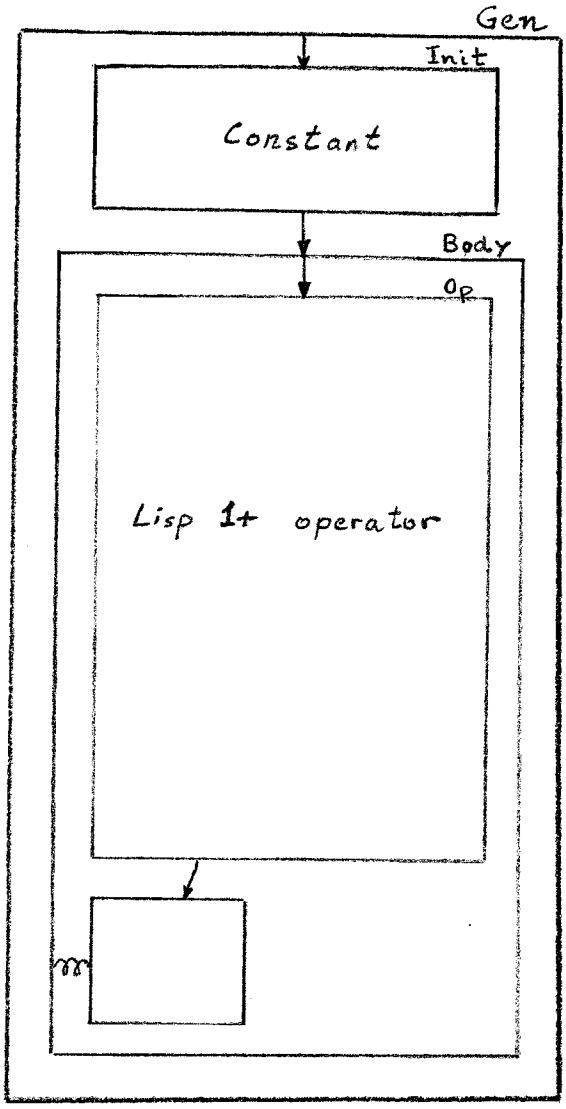


Figure 1: Example of a count generator.

4.4 List enumeration loop

Similarly, list enumeration is the attribute of a loop which enumerates the successive elements of a list until the last one is reached. It must have:

- a generator whose operator gets the next element of the list with the Lisp operator *CDR*,
- a terminator which tests for the end of the list.

This loop is then qualified with the assertion:

```
(loop-list input-list CDR NULL)
```

where `input-list` is the input entity to the generator after initialization. If only the first condition is met, the attribute is called list generator and the corresponding assertion is:

```
(gen-list input-list CDR NIL).
```

Figure 2 shows a partial and simplified plan for the example of 4.2. Only the required data-flow and control-flow links, and operators needed to recognize a list enumerating loop are represented.

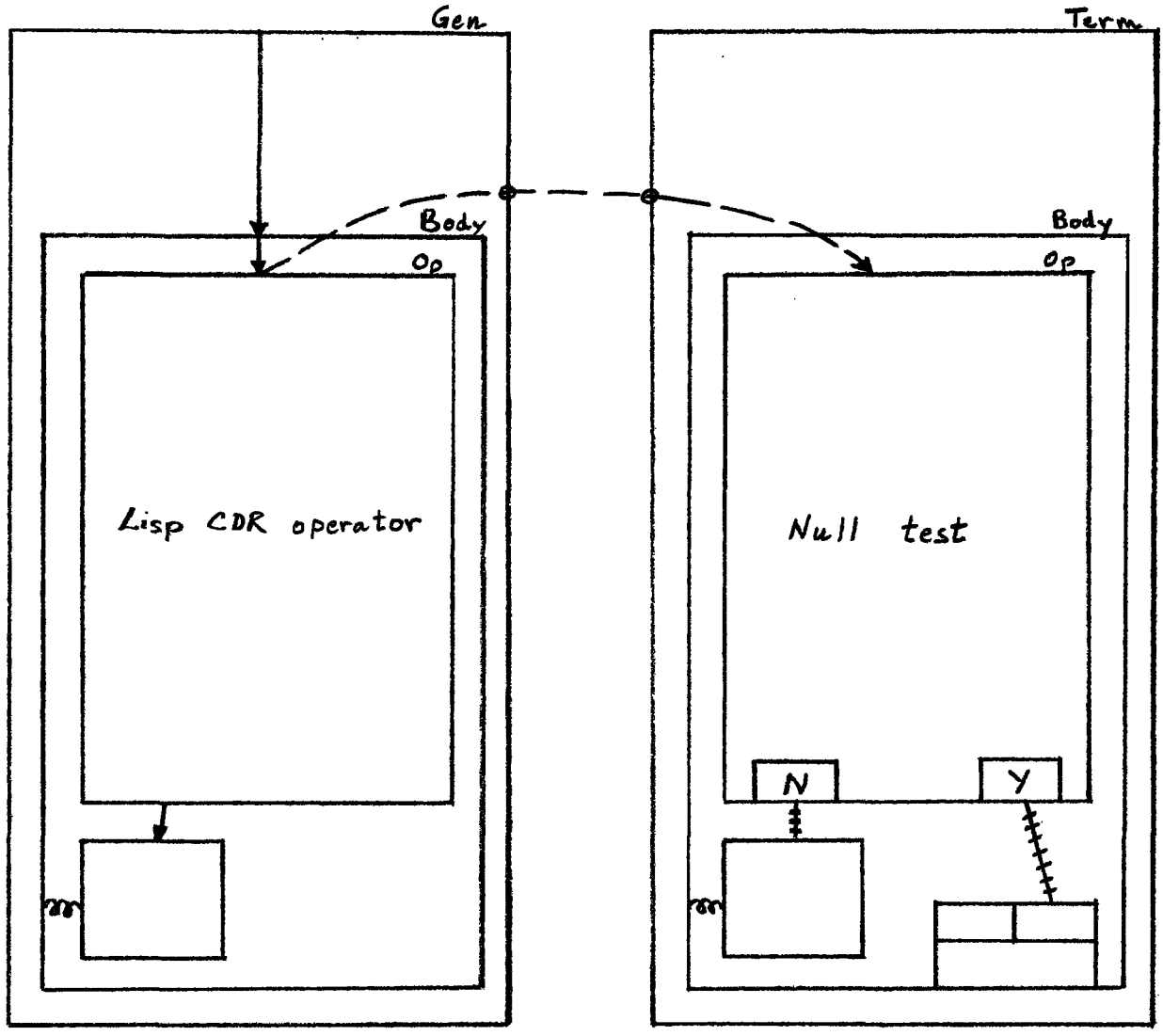


Figure 2: Example of a list enumerating loop.

4.5 Other features

It would be possible to further enhance feature extraction by combining the above four basic features into

higher level features such as:

- standard search loop: this is a count-loop or a list-loop with two exits. One of the exit tests that the count or list generator runs out of elements to test and is the unsuccessful search exit. The other exit must be triggered by a test in which the current counter or list element must be used. Example:

```
(do ((l list (cdr list)))
    ((null l) (return nil))
    (cond ((PRED (car l)) (return (car l)))))
```

PRED is an arbitrarily complex function, but it must use (CAR L) as input.

- another variety of search loop is one where there is only one exit; this happens when one is absolutely sure to find the searched for element and therefore only the successful exit is present.

- matrix traversal: there are two nested count-loops with a data flow between them representing the passing of a column. Vector traversal is an obvious special case.

4.6 Arguments and return types

In order to enhance the comprehension of the English generated text, it was felt beneficial to give as much information as possible on the type of the arguments to a function and the return value. Some cross reference material is also included: indications where the arguments and return value are used or generated.

The generated documentation gives a description of the following four kinds of input and output:
- input arguments to a function,
- the return value of the function,
- input and output free variables i.e. variables which are used as input or output to the function but not formally mentioned in the argument list,
- variables which are subject to side-effects within the function.

In order to determine the type, the effects of all called functions, down to arbitrary levels, are taken into account, in as much as this information is available. This is done by traversing the plans starting at all the input and output ports. For primitive functions, the type of their arguments and result are known to the system and propagated to the ports. An assertion of the following form is generated for each input or output port:

```
(variable indicator (OR (xref type) ... (xref type)))
```

where

- `variable` is the name of the concerned variable,
- `indicator` is either a number which gives the position of this formal argument in the argument list, or takes the value `part` which indicates that the return value is *part* of the `variable`, or is NIL,
- `xref` is the name of a function which uses or generates `variable`
- `type` is the name of a type (e.g. number, vector) implied by some use of the `variable`, or NIL.

Constants are treated in a similar way and their values are stored. If a variable is only used or generated in one place the `OR` is suppressed. Consider the following examples:

```
(tblsize nil (remainder number))
(tbl part (arrayfetch nil))
```

The resulting information is stored as assertions in the plan data base with key words making it possible to distinguish the four different kinds of input and output listed above.

5. Documentation generation

Although it was anticipated that this phase would solely generate English text, it turns out that the information added to the plan is not in a form lending itself directly to documentation generation. The problem is that, although loops do not need to be dwelled on anymore, other control structures - such as conditionals - and expressions need to be summarized, because the plan is still too detailed in these areas.

5.1 Focusing on key features

The main idea for this part is to mark the segments in a plan, indicating whether they should be mentioned in the documentation or not. In fact many segments will be marked not to be mentioned in the documentation, as there are too many of them. The plan is traversed recursively and segments are marked from the bottom up, so that selected leaves are marked first as being unimportant. There is a special procedure defined for each segment type which determines what happens to the segment depending on the nature of the segment and some local context. For instance, for a conditional this procedure marks the segment for not being mentioned in the documentation if the number of its subsegments to be mentioned in the documentation is zero or one: this usually happens when earlier in the scan some of the subsegments of this conditional were already marked unimportant.

5.2 English generation

First, a summary line is generated giving the name of the function, the number of arguments and free variables, and indicating whether the function has side-effects. Example:

```
Function DELETE has 1 input-argument , 2 free-variable-inputs and has side-effects.
```

For each argument, free variable, side-effected variable and the return value, its type when it is known, and the places where it is used are indicated. Example:

```
The input argument is SYMBOL used in CONS and/or HASH.
```

The plan is now traversed a second time, top-down, ignoring all the segments which have been marked unimportant during the focusing on key features. All the other segments are used to generate English text.

5.3 English generation technique

The technique used here is an improvement on the method proposed by Roberts in [3]. His English generation module is a simple generative scheme to build descriptive phrases directly from Lisp function calls. Atoms have a functional property attached to them. When invoked, these functions are recursively evaluated to generate English text. This system has flexibility beyond that of a template system.

Instead of using Lisp function calls, I used the assertions added to the data base during feature extraction and generated English text from functional properties attached to key words of these assertions. The main difficulties encountered with this technique were grammatical agreements and punctuation.[1] In both cases, the deeper problem goes back to the fact that the basic generation algorithm makes no provision for high level grammatical structures such as sentences or paragraphs. At each moment, only the local context is known because the generator does not keep track of the surrounding context.

In order to improve the situation, a number of simple routines were developed which automatically take care of the most used agreement rules in the context I was working in: plural of nouns and verb agreement.

Punctuation is taken care of at a higher level, where the English is generated and global information is available. This is not a very satisfying solution but a workable one. It is unsatisfying because it has poor

---

1. The first problem would have been much worse in other languages such as French or German because agreement rules are more numerous (e.g. adjectives are invariable in English but not in French).

modularity, and it makes it impossible to have a clean interface with the English generation module.

6. Conclusion

The results show the feasibility of this type of approach although a lot more work needs to be done. The documentation generated lacks a certain depth and insight. But it has the distinct advantage that its accuracy depends on an automatic analysis procedure which, among other features, detects many side-effects. Furthermore, because all the processing is done on the plan representation, the work reported here is independent of the user programming language.

Some obvious extensions have already been mentioned in the course of this paper: the recognition of more features such as search-loop, vector and matrix traversing and the use of a more sophisticated English sentence generator.

It is my feeling that the feature extraction is a fairly robust mechanism which can be used on more complex programs. The focusing on key features seems a less robust method and I anticipate that for the analysis of larger and more complex programs, more elaborate heuristics need to be developed.

As mentioned earlier, several levels of detail should be made available with a way for the programmer to specify the desired level. The system should be able to present partial views of the code such as:
- looking only at or ignore completely the error system, the initializations,
- only considering the main control flow without error code and without initialization.

This would allow the programmer to focus more sharply on the problem area he or she is concerned with.

Acknowledgements

This work was made possible only because of the help of Charles Rich and Richard Waters.

Bibliography

[1] Rich C., & Shrobe H.E., "Initial Report on A Lisp Programmer's Apprentice", MIT/AI/TR-354, December 1976.

[2] Rich C., Shrobe H.E.,& Waters R.C.,"Computer Aided Evolutionary Design for Software Engineering", MIT/AIM-506, January, 1979.

[3] Roberts B.,"Building English Explanations from Function Descriptions", MIT/WP-185, April 1979.

[4] Shrobe H.E., Waters R.C.,& Sussman G.J., "A Hypothetical Monolog Illustrating the Knowledge Underlying Program Analysis",MIT/AIM-507, January 1979.

[5] Waters R.C.,"Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, December 1978.

[6] Waters R.C.,"A Method for Analyzing Loop Programs", IEEE Trans. on Software Eng., SE-5, No. 3, May 1979.