A SYSTEM FOR DEVELOPING PROGRAMS BY TRANSFORMATION

by

Martin Stephen Feather

Ph D

University of Edinburgh

1979

CONTENTS

ACKNOWLEDGEMENTS

ABSTRACT

Chapter

1. Introduction

2. The art of programming

    1. Difficulties of programming

        1. Structured programming

        2. Verification

        3. Program maintenance and modifiability

        4. Sidestepping the problem

    2. The potential of program transformation

    3. My own approach towards a transformation system

        1. Underlying method of transforming

        2. Level of transformation

            1. Transformation context

            2. Pattern directed transformation

        3. Control of the system

        4. The use of defaults

3. Review of state of the art

    1. Overview

    2. Martelli

    3. Pepper et al

    4. Manna and Waldinger

    5. Darlington and Burstall

6. Burstall and Darlington

4. User view of system

1. ZAP program transformation system primer

2. ZAP program transformation system users' manual

5. Transforming large examples

1. Transformation tactics

1. Combining tactic

2. Tupling tactic

2. Transformation strategies

3. The telegram problem

1. English specification

2. Design of protoprogram

3. NPL protoprogram

4. Transforming to efficient version

5. Final program

6. Modification of telegram problem

4. Simple compiler

1. Design of protoprogram

2. NPL protoprogram

3. Transforming to efficient version

4. Final program

5. Remarks on transformation examples

6. Transformation of a text formatter

1. Informal specification of the text formatter

2. Design of protoprogram

3. NPL protoprogram

4. Evaluation of text formatting programs

            1. Satisfying informal specification
            2. Resolving ambiguities in the informal specification
            3. Changing/extending the program
        5. Transformation to efficient version
        6. Final program
    7. Implementation
        1. General details
        2. NPL level
        3. Transformation level
            1. Utility section
            2. Control section
            3. Transformation step section
                1. Expansion of expression and pattern
                    1. Normalisation of iterative constructs
                    2. Normalisation of conditionals
                    3. Normalisation of where constructs
                2. Matching of expression and pattern
                3. Instantiating pattern to form answer
            4. Default section
                1. Type information default
                2. Cases default
                3. Pattern default
    8. Conclusions
        1. Summary
            1. Adequacy of techniques and system
            2. Range of applicability
        2. Extensions

1. System improvements

2. Extending transformation methods

3. Comparison with other work

BIBLIOGRAPHY

APPENDIX - NPL

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to the following people and organisations:

To my supervisor, Rod Burstall, for his constant encouragement and assistance throughout;

To John Darlington who, with Rod Burstall, first aroused my interest in program transformation and provided the underlying work which served as my starting point, and has been helpful throughout;

To Dave MacQueen, Alberto Pettorossi, Gordon Plotkin, Robin Popplestone and Jerry Schwarz for useful discussions of transformation and related topics;

To my past and present colleagues of the Department of Artificial Intelligence for making my stay in Edinburgh so enjoyable;

To the Science Research Council for providing financial support and computing facilities.

ABSTRACT

Much of the difficulty of programming can be attributed to the clash between the goal of efficiency and other desirable goals, such as clarity, reliability and modifiability. This thesis proposes program transformation as a suitable methodology for program development to circumvent this difficulty.

Following this methodology, a program is developed by first writing a simple straightforward solution to the problem, unhampered by efficiency considerations. Efficiency is then introduced in a separate step by transforming the simple solution.

In order that this be a practical methodology, transformation of large programs must be possible to perform reliably and easily. This thesis presents an implemented machine-based transformation system which attempts to realise these needs.

The system is based on a concise and powerful transformation method due to Burstall and Darlington. The emphasis of the system is on making it easy for the user to control the system through a transformation. Guidance is expressed in a command language, so that commands may be saved and re-run, modified, or viewed as documentation together with the initial program.

The level at which guidance is given is higher that the low-level underlying manipulations. Techniques for organising the transformation of large programs at even higher levels are presented. Some non trivial programs and their transformation as achieved using the system illustrate these features.

CHAPTER 1

INTRODUCTION

The use of computers is continually increasing, and a great deal of research is being done into the hardware and software aspects of computing. Progress on the hardware side has led to cheaper and more efficient machines, so the cost (in both time and money) of providing and maintaining software is becoming increasingly significant. Since the late 1960's the existence of the so called software crisis has been recognised - that is the difficulty of specifying, developing and maintaining large pieces of software. Consequently there is a great deal of interest in devising methods to ease the task of programming.

My own research has concentrated on one of these potential methods - program transformation. This is a design methodology that suggests we produce a program in a two-stage process; firstly, write a simple program without regard for efficiency of execution (so freeing us to aim for clarity and correctness); the required efficiency is achieved in the second stage, in which we _transform_ the initial program. In making this separation we hope to benefit by ending up with an efficient program (as we would if we used some other design method), yet one which is much more reliable and better documented through being derived from a simple initial program.

Other researchers have already invented ways of transforming programs. Rather than look for yet more such ways, or to extend them to some new domain, my decision has been to take what appeared to be a promising approach, invented by Burstall and Darlington, and attempt to develop it further in the direction of practical applicability by trying it on larger and more complex examples. If transformation is to become a practical methodology, it must be both easy and reliable to perform. This implies the need for a machine-based transformation system to aid us in transforming programs. Such a system would provide reliability, and give assistance by removing the drudgery of carrying out many small operations by hand. Darlington had already developed a semi-automatic system based on the transformation method he invented with Burstall. Although his system performed impressively on small examples, it did not seem to be practical for use on larger programs.

A major part of my work has been to produce my own transformation system which is intended to be a suitable tool for use on larger programs. The system adopts the Burstall-Darlington transformation method as its underlying means of transforming programs, however the transformation steps which the system implements are at a higher level than these underlying operations. Each system step can be justified in terms of many small steps, but the user is saved the need to think at the rather low level of the small steps. An important design decision behind my system has been to accept, in fact encourage, user guidance. As an investigation into the practicality of transformation, I consider it better to see how much can be achieved with the aid of a machine-based system

rather than to see how far a totally automatic approach can be pushed.

To run my system, the user provides a series of commands in a specially designed control language. Such commands may be given at the terminal, or stored in a disc file and executed (or a mixture of both). The commands form a readable account of the transformations carried out, and serve as documentation showing how the initial program is transformed to attain the final, efficient, program.

With the aid of my system I have tackled the transformation of some non-trivial programs. In doing so, the need to structure the transformation process itself has become apparent. To organise a large transformation, I have developed efficiency introducing "tactics", and an overall "strategy" for applying these tactics. In the same way that the transformation steps of my system are at a higher level than the underlying transformation steps upon which they are based, the tactics and strategy can be viewed as acting at a yet higher level in the transformation process.

I have deliberately attempted transformations of programs considerably larger than the examples hitherto tackled. My belief is that an easily guided interactive system is the only way to deal with programs of this scale - a totally automatic approach suffers from the combinatorial explosion of possibilities, and an entirely hand-performed transformation of programs of this size would be too tedious to perform correctly, if at all.

The layout of the remaining chapters of this thesis is as follows:

Chapter 2 - An expansion of my motivation for considering program transformation, and for the design decisions behind my system.

Chapter 3 - An examination of other research into program transformation. A selection of other peoples' work illustrates different approaches to, and uses of, transformation.

Chapter 4 - The instructions on how to make use of my transformation system.

Chapter 5 - The transformation of non-trivial programs. The tactics and strategy I have developed are described. Two non-trivial programs and their transformation as achieved with the aid of my system are presented.

Chapter 6 - The transformation of a text formatter. This program is considerably larger than the examples presented in chapter 5, and I discuss the difficulties which its transformation brought to light.

Chapter 7 - Significant implementation details of my system.

Chapter 8 - Conclusions to be drawn from the work done, and possible avenues for further research.

CHAPTER 2

THE ART OF PROGRAMMING

This chapter examines the task of programming to see why it is hard, then expounds the potential of program transformation as a programming method, and finally presents my own approach to investigating whether this potential can be realised.

2.1 DIFFICULTIES OF PROGRAMMING

Computer programming remains a difficult task requiring much effort and intelligence. Large software projects can require many man-years of work to complete, and with the continuing hardware developments, software costs are becoming the major expense. In order to determine the causes of the difficulty, we must examine the interactions between the goals we seek to satisfy when programming. Typically when writing a program we have some or all of the following objectives in mind:

Correctness - the program should perform correctly the task we intend it to do.

Efficiency - despite the continual reduction in costs of hardware, and increases in performance, programs must still be

reasonably efficient. Often it will not be necessary to achieve the ultimate efficiency possible, but there can still be a large gap between an arbitrary program and a tolerably efficient program.

Clarity - ideally programs should be easy to understand. When this is not so, supplementary documentation is required to further clarify the behaviour of a program.

Modifiability - very often the program we first produce will need to be modified to perform differently later on. Ideally, our initial program should be capable of relatively easy modification when the desired changes in the task are not too drastic.

It is the attempt to simultaneously satisfy these goals that introduces much of the difficulty into programming. Efficiency in particular seems to interact unfavourably with the other goals. In achieving efficiency we usually pay the price of decreased clarity, and risk losing correctness. To achieve efficiency involves combining what we originally conceive of as distinct activities so as to benefit from doing them all at the same time, thus destroying the program's modularity.

A widely-used approach to developing programs is to write them haphazardly, and then eliminate errors by testing. This can hardly be regarded as ideal. We are unlikely to be able to test all paths in the solution, and may well find that some errors remain undetected, only to emerge later when the program is in use and some

unforseen set of circumstances brings them to light. What may seem a clear feature to the programmer at the time of writing may be hard for others, or even that same programmer some time later, to follow.

2.1.1 Structured Programming

Approaches to alleviate some of these difficulties have been developed under the name of "structured programming". Noted texts on this methodology include Dahl, Dijkstra and Hoare [1972], Dijkstra [1976]. Structured programming has two aspects - one is the feature of a structured program: For example, we are encouraged to avoid the arbitrary use of 'goto's, and instead use while-expressions and the like. This does not tell us how to write structured programs, only what features tend to make programs unstructured.

The other aspect is the orderly development of a program - i.e. structured program<u>ming</u>. I consider stepwise refinement, and data abstraction. Stepwise refinement is the development of a program from a specification of the problem by a progression of stages, each going into more detail than its predecessor. Calls to procedures are written first, before their bodies - in doing so clarifying what the procedures are to do. Whilst this is better than an entirely haphazard development, we are not guaranteed to avoid introduction of errors during the development. Also, we are expected to make appropriate choices at each level of refinement. As we descend into more detailed levels, the earlier decisions dictate constraints which cannot be changed without going all the way back to them and re-doing the development from there. A feature of stepwise refinement which

will be seen to contrast with transformation is the way in which the overall structure of the final algorithm is fixed from the very start of the development process. The process is, as its name suggests, a refinement, descending into more detail but not carrying out any major structural changes. Thus the approximate structure of the final efficient algorithm must be present during all stages of design. For purposes of clarity, modifiability and verification we would like to deal with program structures not encumbered by the additional complexities of efficiency. Stepwise refinement is of no assistance in making major structural changes that incorporating efficiency into a naive algorithm would require.

Data abstraction is the development of a program by building the algorithm around the data structures appropriate to the problem, and the operations we require upon them. This provides a form of modularity. We make a distinction between the operations upon the abstract types, which we use throughout the program, and the actual representation of the types and implementation of the operations in terms of features actually available in the language. This implementation is hidden from the rest of the program. It gives us the security of knowing that the objects we build up are well formed. The modularity allows us to change the representation by only having to consider the part devoted to representation, the calls to abstract operations throughout the remainder of the program remaining unchanged.

Unfortunately efficiency may force us to abandon our good ideals once again. We are tempted to relax the barriers hiding representations so as to be able to make computational short cuts

based upon our knowledge of the representation in use. For example, we may be manipulating sets of objects. Our representation for sets may be such that for a given set, selecting elements from it always produces them in the same order. We might take advantage of this property in the main body of the program, which could lead to trouble if we were to change the representation later so that the property no longer held.

2.1.2 Verification

If one of our fundamental goals is correctness, we may be prepared to put a lot of effort into developing a program and then proving it correct with respect to some specification. Although this does not aid modifiability or clarity, there may be occasions when these are not regarded as essential - typically when we wish to generate a "service" program which is to be used often, and must perform faultlessly.

Program proving has developed from the early work of McCarthy [1963], Floyd [1967], Manna [1969], Hoare [1969] and Burstall [1969]. The most intricate verifications have been performed by hand, requiring considerable insights into the problems. To combine the reliabilty of a machine based system with the intuition of a human, work has been done to develop interactive verification systems (e.g. Good [1970], Topor [1975]). Fully automatic verification systems have had their greatest success on rather limited domains of programs. Functional languages are seemingly easier to prove properties about than conventional imperative languages. Work done by Boyer and Moore [1975], and Aubin [1976] demonstrates this. Despite the considerable

attention this area has received, there have been no great breakthroughs -- the main result to emerge has been the realisation that program verification is hard.

Sometimes verification has the beneficial side effect of providing insights into the behaviour of a program. For example, assertional methods of verification can give rise to invariants which are instructive about performance. To follow the whole of a verification in order to understand the program may itself be rather tedious. Indeed, for non-trivial programs, the entire verification may be so lengthy that to follow through all of it would be of no value in convincing us of its validity. This is one of the considerations behind the implementation of LCF (Milner [1972], and Gordon, Milner and Wadsworth [1976]), which is designed to allow us to make proof generators, taking us away from the detailed level of the proof itself, and making the overall proof generation and comprehension much easier (Cohn [1979]).

Despite this sort of advance, as a method of understanding programs verification is not ideal. The intertwining of algorithmic details with efficiency details means we must try to comprehend details relating to both at once, instead of being able to break the problem down.

Verification presupposes that we can produce an acceptable specification. not always a trivial operation. Programs whose description are given loosely in English, but for which no simple formal specification are apparent are particularly hard to specify. (How, for example, does one concisely specify a text formatting program ?). In writing the program we must resolve ambiguities which may be present in the informal presentation. The danger is that from

the final program alone it might not be clear how the ambiguities have been resolved - worse still, we may never have noticed the existence of some of them, and unwittingly made choices which turn out not to be the best. An illustration of this is the Telegram Problem, originally presented by Henderson and Snowdon [1972] as an example of structured programming not preventing the introduction of errors. The specification of this problem is given in English, and as such is incomplete. This is one of my examples in Chapter 5, so I reserve further discussion of this until then.

2.1.3 Program Maintenance And Modifiability

In practice a considerable amount of effort is devoted to program maintenance and modification. Once a program has been written it may need modifying because its behaviour is not as expected, or the desired behaviour - i.e. the specification - changes. Indeed, for many applications we may expect our specification will change in the future, but at the time of initial design cannot predict what these changes are to be. Making changes in programs designed for efficiency is extremely hard to do correctly. An adjustment to make one change may introduce other undesirable changes in the process. Certainly the clearer and better modularised a program is, the easier it will be to see how and where to make a required change. Unfortunately even if our efficient program is clear and well modularised, incorporating a succession of ad hoc changes will break down modularisation, making further changes increasingly hard to perform. This is observable in practice in the development and maintenance of large programs, when after a certain

point it becomes worthwhile to re-write the entire program completely rather than continue attempting to make changes to what has degenerated into an unstructured mess. Structured programming techniques do not help us make changes in programs whilst preserving structuredness.

2.1.4 Sidestepping The Problem

We see that we face the task of attaining several incompatible goals. Fundamental to their incompatibility is the clash between good structure and efficiency, both of which we desire. Because of this, any approach to programming intended to produce just a single program satisfying all our goals seems doomed to failure.

I now consider methodologies designed to get around this problem:

An interesting approach that has received only a small amount of attention is to describe a program in two parts. One part is a set of recursion equations, predicate logic, or some similar presentation of the basic description of the program. The other part is a set of annotations which further indicate how the equations, logic, or whatever, are to be used to calculate the results.

We imagine the existence of a compiler which accepts both of these parts, and runs the program in the indicated fashion. The advantages of this approach stem from the additional modularity we get from the separation of the basic description (the "what" part) from the operational aspects given in the annotations (the "how" part). This modularity helps improve the simplicity and clarity of

the whole, thus making it potentially easier to understand, verify and modify, yet combined with the compiler the running program need be hardly less efficient than a program we would have developed conventionally. Work of this nature has been done by Hayes [1973], Kowalski [1977], Schwarz [1977] and Warren [1977].

This approach may be unable to achieve quite the efficiency that code compiled specially for the problem can. Also, it is not clear how large a set of annotations, or whatever are used to specify the "how" part, will suffice for most of the behaviours we are likely to want. There is scope for further investigation in this area, but this is not the direction I have chosen to investigate.

The approach I would like to consider in some detail is program transformation.

2.2 THE POTENTIAL OF PROGRAM TRANSFORMATION

The "transformational" approach to programming suggests that we develop programs by first ignoring efficiency aspects, writing the clearest, most straightforward program possible to perform the task. Then, as a separate process, transform this into a sufficiently efficient version.

In the real world of commercial programming, M. Jackson [1975] has done much to highlight the fundamental difficulties in program design, and his techniques for construction of programs are related to the fundamental issues behind the transformational methodology.

By adopting this approach we do not have a single program which we examine for each of our criteria, instead we have two programs -

the first simple program, which I call a PROTOPROGRAM, will serve as our precise specification. Because its design is unhampered by efficiency considerations, it will reflect only what we want done, without confusing us with the precise details of how it should be done. This gives us the scope to use any techniques we wish to produce a clear program. We are now free to make use of functional languages, based on expressions and recursion. Assignment can be disallowed, since it is a major source of error introduced on efficiency grounds. We can tailor data types to our requirements rather than to the implementation. High-level constructions related to our data types can be permitted - for example, if we were dealing with sets, we would want to converse in set expressions of the form { f(x) : x in S and p(x) } instead of having to explicitly construct looping or recursion over set S. Modularity and data abstraction can be used to divide large programs into self-contained pieces.

If we wish to prove correctness properties, it will be easier to do so on the protoprogram than when efficiency has been incorporated. Ambiguities in the informal specification will be resolved in the design of the protoprogram - which we are much more likely to write to perform as we desire - and the choices made will be readily determinable from this later.

Almost always our protoprogram will be unsuitable for practical use. The transformation process aims to convert the protoprogram into an equivalent, but much more efficient, version. Provided the transformations preserve correctness, the final program will be as correct as the initial one. The documentation for the final program is the protoprogram together with a description of the transformation

steps applied to it. Further, each function of the final program has its effect expressed in terms of the functions in the protoprogram.

Modification can now take two different forms. The first is when the protoprogram remains unchanged, but we need to adjust the transformations in order to direct them towards changed efficiency criteria. Sorting problems illustrate this feature - for example, if for some domain comparisons are "cheap", our efficient algorithm might perform many of them, reducing the number of exchanges between items. If, however, we wish to sort in a different domain where comparisons are expensive, we would want to change the transformations to head towards an efficient program which minimises comparisons rather than exchanges.

The alternative form of modification concerns changes to the protoprogram, i.e changes in specification. The simplicity of the protoprogram should permit changes to be incorporated easily and correctly. This contrasts with the difficulty inherent in altering efficient code - because this tends to be very intertwined, even small changes may have far reaching and hard to determine repercussions. Our crucial step is how the transformation of the modified protoprogram goes through. Our hope is that the original transformation will not require much adjustment, and that detecting where changes might be necessary will be easy. If this is the case, we have avoided the need to re-do all the transformation work, and reliable modification will not imply excessive amounts of effort. Of course, for some changes the transformation will require very significant adjustment, leading to a very different final program. In such cases it would almost certainly have been impractical to

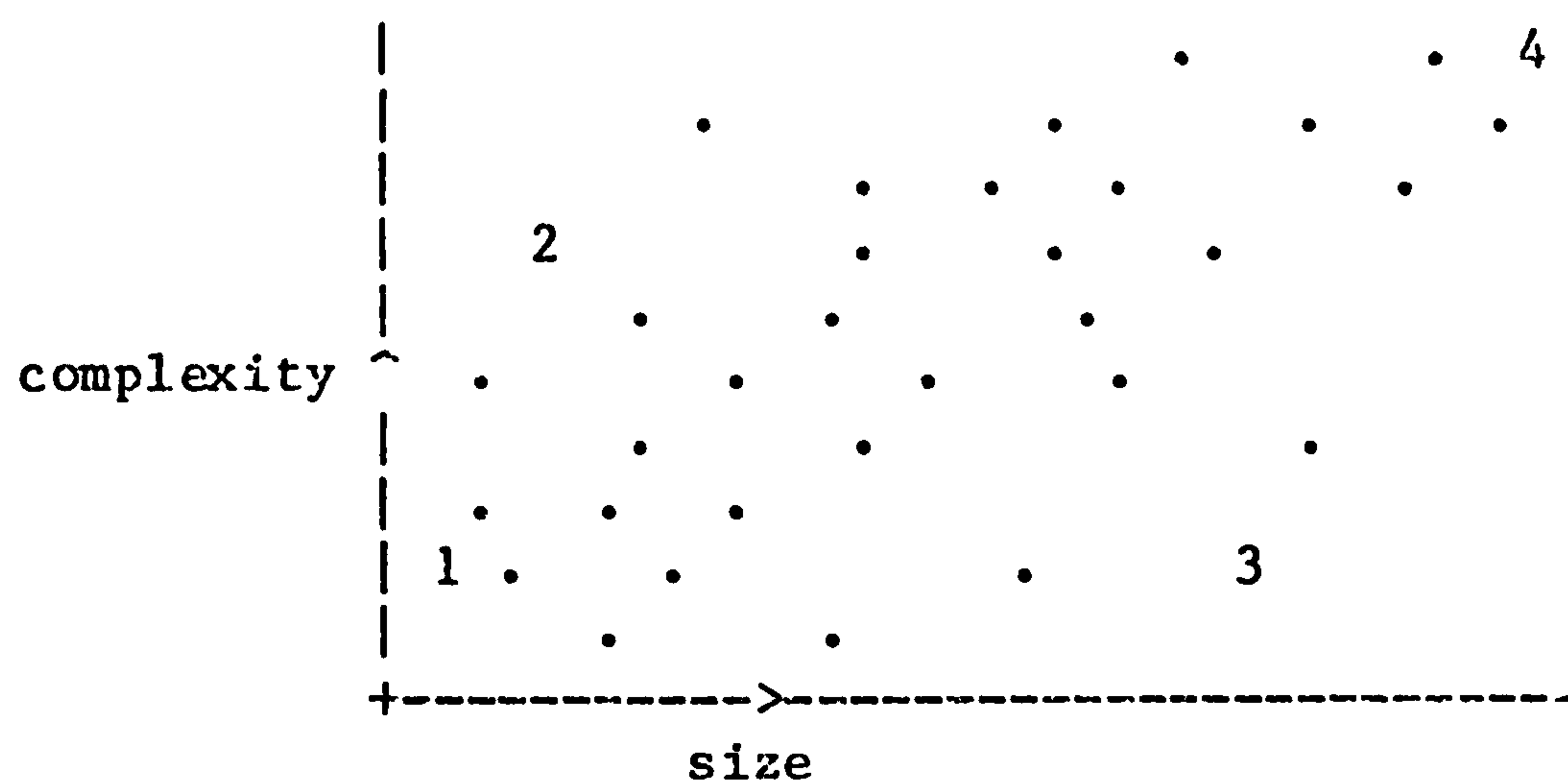
modify the first efficient program, although perhaps this would not be obvious until we had expended some futile effort trying to do so.

Programming purely for clarity, discarding all thoughts of efficiency, can be suprisingly hard for someone familiar with conventional styles of programming. We tend to perform mental optimisations when writing programs. The danger is that what may seem at the time a simple optimisation can later seem rather hard, and we then regret having attempted to take this short cut on efficiency grounds. This is a habit which must be overcome if we are to make the most of this approach to programming.

Compilers can be viewed as very straightforward, totally automatic, transformers, converting from a source language in which we can write and think more easily about programs, to machine code for executing them. We sacrifice some efficiency in exchange for the high level language. Notice however that the source code we write closely influences the behaviour of the machine. Even in optimising compilers the algorithmic changes are very low level (such as removal of redundant calculations from loops). The improvements I am concerned with here are of a much more sweeping nature, involving overall modification of the algorithms used.

Thus program transformation has the potential to be an extremely appropriate method of developing programs. Whether this potential can be realised depends upon how easy or otherwise it will be to carry out the transformation itself. If this turns out to be a long and difficult activity, we lose the advantages we hoped for. A long, hard to follow set of transformations does not serve as documentation, would be hard to modify, and could well contain

errors. From the state of the art survey (Chapter 3) we see that most of the work done so far has been on only very small problems, for which it has been a trivial operation to write the final program straight off, so these alone must not be regarded as a guarantee that the method is appropriate for "real life" problems. We need to assertain how the difficulty and length of the transformation increases with the difficulty and length of the initial program. Intuitively we see there are some problems which are intrinsically complicated without being particularly large (e.g. Dijkstra's on-the-fly garbage collector, Dijkstra [1976a]) and at the other extreme, large but simple problems (e.g. a payroll program). Worst of all are those which are both complicated and large (e.g. operating systems). Schematically, the distribution probably looks like:



1 - trivial problems
2 - complex but small
3 - large but simple
4 - large and complex

Region 1 is as far as earlier machine-based transformation systems have progressed. This thesis is an attempt to push into region 3. Transformation work must be extended further into non-trivial problems to give us an indication of how it will behave. As the problems become larger but no more complicated we would hope that the

difficulty of the transformation would not increase excessively - only its length. We might expect a situation analagous to that in program verification, where the complete proof, whilst not having much intellectual content, is so lengthy as to be practically unintelligible by humans.

This and other considerations lead us to suggest a machine based transformation system. The advantages of this would include .

Reliability - the system would not make mistakes in performing transformations. Provided our underlying method of transformation was valid, we would be assured of maintaining correctness.

Book-keeping - the relatively boring and repetitive tasks in the transformation could be left to the system, thus greatly easing the burden on us.

Discovery - the system might be able to assist in the transformation process, suggesting alternatives to the user running it, and/or filling in details when following a user-provided suggestion.

Control - we might be able to issue commands to the system at a level above that of the basic transformation steps, thus overcoming the difficulties associated with the sheer length of the solution.

Making the assumption that a machine-based system _is_ desirable, we still have choices to make. Firstly, we must decide what the basic steps of the transformation are to be. Since the correctness of the entire transformation will depend upon the correctness of the

individual steps, it might be convenient if there were only a limited number of them, rather than allowing additional ones to be introduced, each requiring verification. At the same time we must decide what language we will write the initial program in, and between what languages the transformations act.

Secondly, we have to determine to what extent the system will behave automatically. There is a trade-off between the need to interrogate the user for guidance and the waste involved if the system futilely goes down blind alleys rather than seek such advice.

At present it would be rash to claim that some particular approach is the best - we must try those that appear plausible, and see whether they confirm that program transformation could achieve its potential as a good approach to developing programs.

2.3 MY OWN APPROACH TOWARDS A TRANSFORMATION SYSTEM

My two main objectives have been to:

explore further into the region of non-trivial problems;

develop a system capable of use by people not familiar with its implementation.

With these in mind, the choices I made were as follows:

2.3.1 Underlying Method Of Transforming

I chose the fold/unfold method developed by Burstall and Darlington to be the backbone of my system. This was partly because, at the time, both Burstall and Darlington were here at Edinburgh, and

their programs were available on the computer, so the bottom level of a system was already implemented. Burstall had implemented an interpreter for a simple functional language, NPL, embodying features encouraging clear programming style. Also, Darlington's system performed convincingly on small examples, and done-by-hand investigations indicated the basic approach promised to extend to somewhat larger examples. For a description of their work, see Chapter 3.

It is important to note that the fold/unfold steps work on a simple recursive language, so any final program we get from their use will remain in this language.In particular, the language is purely applicative, having no form of side effects. Consequently there will be an unavoidable overhead if we remain in such a language when we need to modify part of a large data structure, since it will require the complete reconstruction of that structure. To convert to a more conventional iterative language as the last stage in a transformation is outside the scope of these rules. This conversion will not change the program structure, and it is the ability to make very major structural changes from protoprogram to efficient program that we benefit from. The question of converting applicative style programs to make use of destructive operations is a field I have chosen not to enter, but one in which work is required. Research has been done by Pettorossi [1978] on how to introduce destructive operations so as to improve memory utilization whilst preserving correctness, and by Schwarz [1978] on means of verifying that uses of destructive operations within a program still preserve correctness.

2.3.2 Level Of Transformation

Darlington's system operates at the level of the fold/unfold operations. The user's responsibilities are consequently also at this level - he sets switches to control folding, and accepts or rejects individual folds.

The effects of working at this level are to make tackling small examples easy, the user provides guidance by a small amount of switch setting and answering yes/no questions asked by the system. Provided the system need not do many folds (and does not have a large choice of folds), such guidance will be easy to give.
Large examples are much harder, however. The unfold/fold steps become noticably too small, there being many possible folds involved in each transformation. The user is rapidly overwhelmed by the many questions the system asks - particularly when the switches have been set to cause the system to act in its most powerful manner.

Since one of my aims is to attempt larger examples, my system must operate at a higher level than individual fold/unfold steps. I achieve this by defining a context in which transformations are carried out, and by introducing a new way of guiding transformation, which I call pattern directed transformation.

2.3.2.1 Transformation Context - When tackling large problems, any particular transformation will typically involve only a small part of the entire program. Defining a context for a transformation limits attention to only the parts which will be required.

Within a context the following will be indicated:

equations to be used for folding and unfolding

lemmas to be used during unfolding

functions which may occur within the answer sought

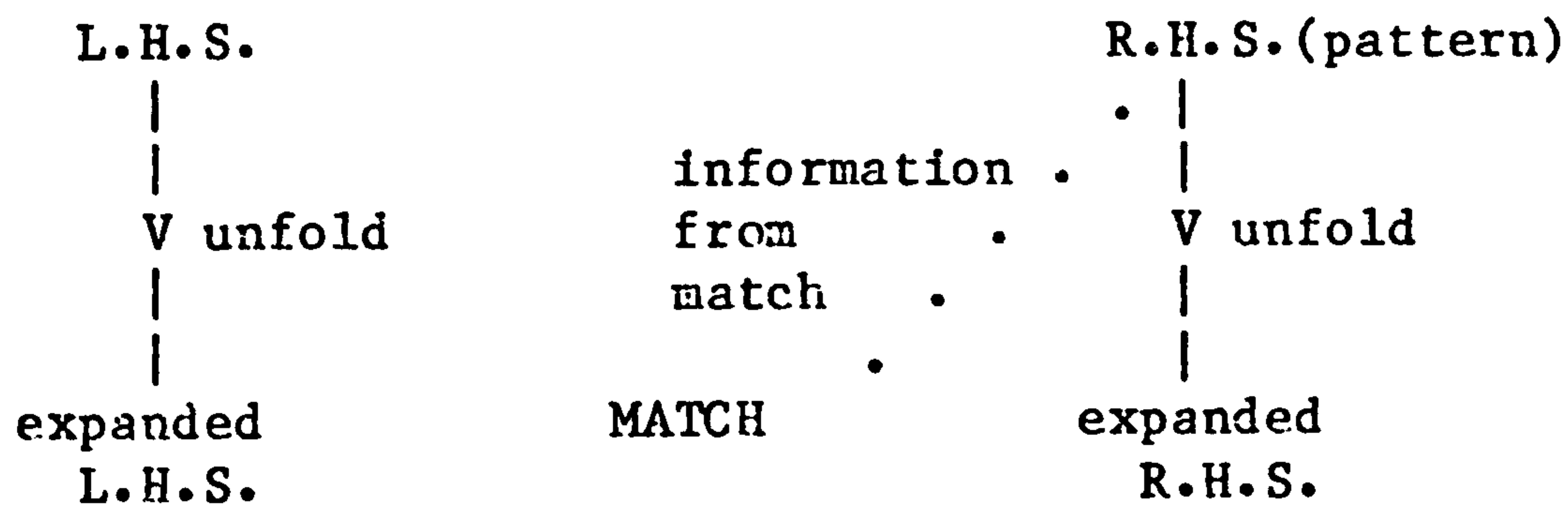
2.3.2.2 Pattern Directed Transformation - This is the means by which an individual transformation is indicated. The user gives the approximate form of the answer he expects in the form of a pattern (so called because it contains variables which will be used in matching). The transformation process becomes:

expanding the expression to be evaluated (unfolding using equations, applying reductions whenever possible);

expanding the pattern;

matching the expanded expression and pattern, and if successful, using the bindings formed by the match to instantiate variables within the original pattern to give the answer.

Pictorially



The advantage of this approach is that the single step of giving a pattern may cause a transformation which is based on very many fold/unfold steps. Pattern directed transformation forms the higher level at which transformations are carried out within my system.

The price paid for moving to this higher level is the need to

give a pattern representing the approximate shape of the answer sought. This turns out to be only a small price, however. For simple transformations only very simple patterns will be required, and my implementation is able to generate such simple patterns itself if so requested. For more complex transformations the power of this approach rests in the ability to express only an approximation of the answer sought - the details of how to do this I leave until the description of how to use my system, Chapter 4.

2.3.3 Control Of The System

I intend that the system should provide an overall service to the user - that is, not only permit transformations to be carried out, but also ease the users task of introducing, testing or saving initial, intermediate or final versions of programs. This becomes of use on larger examples, when the transformation is no longer the simple matter of making a single conversion from initial to final version, but is split into several stages.

To control the operation of all aspects of the system I have developed a control language and documented this (Chapter 4). I had in mind users not familiar with the intricacies of the implementation who would want to make use of the system (not the case with Darlington's system which is primarily a research tool). The control statements serve as a readily comprehensible record of the transformations carried out when tackling a problem.

2.3.4 The Use Of Defaults

The application of the system to perform simple transformations is eased by incorporating defaults which the user may direct the system to apply. In the earlier section on pattern directed transformation it was mentioned that the system is able to generate simple patterns itself. This is one form of default. The other form is concerned with making use of the data types of programs written in NPL to generate simple "type information". Type information is used to split the transformation of a function into several cases (by considering cases of the argument(s) of that function), and is used in the generation of default patterns.

The approach of providing defaults for the user to apply when he thinks appropriate contrasts with the heuristic strategies built in to Darlington's system. The latter involve the user in making a few initial decisions, after which the system goes ahead applying the heuristics to perform the entire transformation. Again it seems that Darlington's approach works well on small problems, but not on larger ones. With larger problems the degree of difficulty of individual transformations may vary enormously, and a heuristic powerful enough to cope with the harder transformations (if such a heuristic exists), will be unnecessarily powerful for the simpler ones. If the user is in control, however, he can use defaults for the simpler transformations, and guide the system through the harder ones himself.

CHAPTER 3

REVIEW OF THE STATE OF THE ART

In this chapter I review and contrast other peoples' work in the field of program synthesis and transformation.

3.1 OVERVIEW

The underlying features characterising the different approaches to program transformation and synthesis are as follows:

The rules for manipulating programs - at one extreme <u>any</u> change in a program which can be verified might serve as a valid transformation step. At the other extreme there may be a fixed set of small manipulations which can be repeatedly applied to achieve large transformations.

Degree of automation - at one extreme transformations may be performed by hand. At the other a machine based system attempts transformations entirely automatically. Between these extremes lies the approach of using a machine based system, but relying to some extent on human guidance. Clearly the degree of automation will influence the complexity of transformations which can be attempted. A system making use of human guidance would be able to achieve much more than a fully automatic system. Likewise, hand-performed transformations might be hard to carry out even on user-assisted

systems. However, we may have more confidence in the transformations performed on a machine based system, and long but not particularly complex transformations may be too unwieldy to do entirely by hand.

Domain of transformations - this concerns the start and end points of the transformation or synthesis. The distinction between transformation and synthesis is that the former begins with some executable (but perhaps intolerably inefficient) program and aims to improve efficiency, whereas the latter starts with some non-executable specification, and derives an executable program from it. The rules for manipulating programs will restrict the domain of the transformation. Some rules are unable to deal with programs involving assignment or side effects, hence the end point of a transformation using these will require conversion into a conventional form to take advantage of such effects.

In the following sections I illustrate different approaches to transformation and synthesis by presenting work of other researchers in this area.

3.2 Martelli

3.3 Bauer et al

3.4 Manna and Waldinger

3.5 Darlington and Burstall

3.6 Burstall and Darlington

Briefly, the nature of their work in light of the characteristics of transformation is as follows:

3.2 Martelli - a hand performed transformation of a complex algorithm

3.3 Bauer et al - a proposed machine based system relying

entirely on user guidance for developing programs

3.4 Manna and Waldinger - a totally automatic program synthesis system

3.5 Darlington and Burstall - a semi-automatic transformation system, applying schemata to perform certain classes of improvements in programs

3.6 Burstall and Darlington - a semi-automatic system applying a small set of rules to perform synthesis and transformation on recursion equations

3.2 MARTELLI

References: Martelli [1978]

In this paper Martelli applies done-by-hand transformations to a non-trivial algorithm. The problem he considers is to copy cyclic data structures. The initial program is a simple recursive solution to the problem, and the derived final program is a realistic algorithm requiring only bounded workspace and linear time.

His objectives are to demonstrate the correctness of the final program by proving that the transformations preserve equivalence, and also to obtain a better understanding of the efficient version by seeing how it can be derived.

All versions of the program are written in PASCAL. The transformations between them are quite complex, requiring a good deal of ingenuity, as do the proofs that they retain equivalence. The transformation steps are motivated to deal with one aspect of the problem at a time, and it is this which permits a clear understanding of the whole process.

This work illustrates how an entirely hand-performed transformation can change a simple program into a highly efficient one. The steps of the transformation are not systematised, hence they can be as powerful as necessary, but require verification. To perform a transformation on a similar algorithm would require as much effort again, and it would be very hard to carry out such a transformation on any existing machine-based system.

Such hand performed transformations serve as aids to understanding and verifying complex programs. Particularly interesting is the investigation of classes of algorithms resulting from the alternate ways of transforming a single initial specification. Work of this nature includes Darlington [1976a], in the area of sorting algorithms, Schmitz [1978], in the area of transitive closure algorithms, and Gerhart, Lee and deRoever [1979], in the area of list copying algorithms.

3.3 BAUER ET AL

References : Bauer, Broy, Partsch, Pepper and Wossner [1978]. Bauer, Partsch, Pepper and Wossner [1977]. Broy [1977], [1978]. Gnatz [1977]. Gnatz and Pepper [1977]. Kreig-Bruckner [1978]. Partsch and Pepper [1976], [1977].

The objective of this group is to develop what they call a system for "computer aided, intuition guided programming". By "computer aided" they mean the system is designed to take from the programmer the burden of clerical work. This includes production of new program versions from old ones by application of formal rules,

preservation of all versions, and documentation of the development history. By "intuition guided" they mean it will be up to the user to direct the whole development process. They do not seek to introduce heuristics to control automatic program development.

Their system has not yet been implemented, but they have already investigated, by hand, examples to help decide what features are required. The kernel of the system is to be an extendable catalogue of transformations (presumably all proved correct), much akin to the early schema work of Darlington and Burstall. Essentially a transformation consists of two program schemata, the "input template", and the "output template" together with preconditions to be satisfied before the transformation will be applied. Sometimesd it is also necessary to specify the location, within an actual program, where a transformation is to be applied.

The transformations are intended to cover the following areas:

introduction of user-provided definitions

manipulation of functional procedures

defining language extensions

manipulating iterative programs

changing between recursive and iterative versions

The user may use the system to introduce his own transformations - in which case he has the responsibility of ensuring their correctness.

Control of the system is carried out by the user, who selects an appropriate transformation to be attempted at each step, perhaps backtracking if a blind alley has been entered, until a suitable version of the program has been reached. There is deliberately no

incorporation of strategy into the system, which acts as a "slave" to release the user from clerical work and risk of careless error.

Attention has been paid to the book-keeping side of the system, which must fulfill certain tasks in a reasonably economical manner. Typical tasks are:

go back to an earlier version (backtracking)

continue with an earlier version (independent development of individual parts)

go back or continue with an earlier version which fulfills some specific condition

record some transformation

show the current stage of development (of the whole program or of parts of it)

Since they started some time after the earliest work in this field, they have been able to incorporate many of the discoveries into their plans. Their domain is wide ranging - all the way from high-level specifications down to assembler code.

Examples they present in the referenced papers are:

Tower of Hanoi (recursive to iterative)

Fibonacci numbers (recursive to iterative)

fusc - a numeric function akin to Fibonacci (recursive to iterative)

Chinese Rings problem (recursive to iterative)

Gray-Code generation (iterative to efficient assembler)

Their proposed system relies upon a large and extensible set of rules to modify programs. This gives their system the scope to tackle a very wide range of problems. Two important points remain to be made.

Firstly, their system is not yet implememnted. The proposals seem well thought out, but it remains to be seen if they can be sucessfully implemented, and how well their system will perform in practice.

Secondly, their decision has been to rest the burden of guidance of the system entirely on the user. This may prove to be an excessive burden when attempting large transformations involving many steps. Again, this can only be finally determined when the system has been implemented.

3.4 MANNA AND WALDINGER

References: Manna and Waldinger [1975], [1977], [1977a], Waldinger [1977]

The authors are involved in investigating and implementing techniques for deriving programs systematically from given specifications. Their approach is to transform specifications by repeated application of rules until a satisfactory program is produced. Specifications are presented in predicate logic. The target-language is LISP-like.

They represent knowledge in the form of many rules within the system. The knowledge is about the subject domain, numbers, lists, sets, etc., meaning of constructs in the specification and target

languages; basic programming principles.

The synthesis process is driven by attempts to achieve goals. The rules encoding programming principles attempt to satisfy these goals, so deriving an executable program. Rules of this nature include:

Conditional formation rule: when attempting to prove or disprove some subgoal of the form prove P, introduce a case analysis and consider separately the cases in which P is true and P is false. This causes the introduction of a conditional expression into the program being synthesised.

Recursion-formation rule: if, in attempting to achieve some goal we need to achieve a subgoal which is a precise instance of that original goal, try to achieve this by expressing the subgoal as a recursive case of the outer goal. This leads us to introduce recursive calls within our programs - but it is necessary to ensure termination when introducing such calls. This rule is equivalent to the "fold" rule of Burstall and Darlington (see last section of this chapter), and the two groups discovered their rule independently at about the same time.

Procedure-generalization principle: this principle suggests that if, when tackling some goal, we are led to achieve a subgoal which is not precisely an instance of the original goal, then generalise to get a goal for which both are instances. This causes an attempt to synthesise a more general program. The authors relate this to theorem-proving work, where it is often necessary, in proving a theorem by mathematical induction, to prove a more general theorem, so that the inductive hypothesis will be strong enough to allow the proof of the inductive step to succeed. The recursion-formation rule

turns out to be a degenerate case of this rule.

In addition to the the usual synthesis from specification to recursive program, they have also used their techniques to produce straight-line structure-changing (i.e. with side effects) programs. This latter feature has been implemented in a system of its own (see Waldinger [1977]). Most of the synthesis work has been implemented in their DEDALUS system, designed to be fully automatic in its operation. The only controls they provide over selection of an appropriate rule from several possible candidates are to (possibly) attach some extra condition to rules to limit their application (which could be used, for example, to prevent a rule from being repeatedly applied to the subexpressions it produces), and to have a preference ordering between rules.

The implementation incorporates the principles of conditional formation, recursion formation, and the special case of procedure generalization in which a new procedure may be formed but no generalization is required. DEDALUS is able to produce termination proofs for recursive programs which do not involve mutual recursion. Representative programs constructed by DEDALUS are:

The subtractive, Euclidian and binary greatest common divisor algorithms.

The remainder from dividing two integers.

Finding the maximum element of a list.

Testing if a list is ordered.

Testing if a number is less than every element of a list of numbers.

Testing if every element of one list of numbers is less than every element of another.

union, intersection, membership, subset and cartesian product of sets.

The methods they have derived for synthesis seem quite powerful, however their implementation of these into an automatic system lags behind somewhat. If the full power of the generalization technique is to be included, controls over it will need to be created. Certainly better strategies for selection of appropriate rules will be required. If they wish to continue providing termination proofs, it would be nice to see their automatic provision of these extended to handle mutual recursion. The characterising feature of their work is the intention that their system be totally automatic. This severely limits the size of problems they are able to tackle, and the indication is that as they attempt larger problems, the combinatorial explosion of possibilities will force their system into excessive searching.

Synthesis work has also been done by Green et al, [1975], [1976] and [1977]. Wegbreit [1976] considers how analysis of program performance can highlight areas for improvement, and help guide the transformations to achieve this improvement.

3.5 DARLINGTON AND BURSTALL

References: Darlington [1972], Darlington and Burstall [1976]

The earlier work of Darlington and Burstall was a mainly schema driven method of converting programs written in a non-imperative language of recursive definitions into an imperative language. The transformations were carried out (largely automatically) by a machine

based system, which could be guided to try one of four types of improvement:

Recursion removal

Eliminating redundant computation by merging common subexpressions and combining loops

Replacing procedure calls by their bodies

Causing the program to re-use data cells which are no longer needed

Their objectives were to develop transformations to improve the efficiency of programs and implement these in a system which would act as an assistant to the programmer, allowing him to program in a lucid style and use the system to help derive an efficient final program.

Transformations were carried out using built-in rules consisting of a recursive schema, an iterative schema, and conditions to be satisfied to ensure the iterative schema was equivalent to the recursive one. With a small amount of user control, the system was able to perform conversions by matching the initial solution to the schema and conditions of the indicated transformation, instantiating the iterative schema to get the final result if there was a successful match.

This method could tackle relatively complex examples provided they fitted one of the provided transformations. The transformations were not complete, and there was no provision for the user to extend them.

Darlington's work typifies the use of sets of complex transformations to manipulate programs. This approach has received attention of others since then, including Standish et al, [1976] and

[1976a], and Loveman [1977].

Kibler as part of his thesis work (Kibler [1978]), has developed a system called SPECIALIST which performs a limited type of optimisation on programs in an Algol-like language. The basis of his system is a set of 50 transformations, in fact small schemata, which are applied by the system with a minimal amount of user guidance. His system accepts a program together with a constraint on its input data structure, and simplifies the program to take advantage of that constraint, but makes no attempt to modify the algorithms involved.

3.6 BURSTALL AND DARLINGTON

References : Burstall, R. and Darlington, J. [1977], Clark, K. and Darlington, J. [1977], Darlington, J. [1975], [1976], [1976a], [1977] and [1978].

With the experience they had gained from their earlier work, Burstall and Darlington were led to further consider improvements to be made to programs in recursion equation form. Influenced by Boyer and Moore's [1975] program for proving facts about LISP programs, they adopted the view that as much manipulation as possible should be performed before removing recursion. They developed a set of six transformation rules on recursive equations which formed an elegant yet powerful method to manipulate such programs. Darlington was responsible for the development of a machine based system to carry out transformation and synthesis using these rules. Burstall developed a simple recursion equation language - NPL - for which he wrote a type checker, parser and interpreter. This language was

inspired by the transformation work, and was designed to provide a vehicle for encouraging a clear, lucid programming style. Darlington adopted a slightly restricted subset of NPL as the input to his system.

This small set of rules proved to be both flexible and powerful. The areas Darlington has applied it to are:

Synthesis - by providing reduction rules for new constructs, definitions using them can be converted into orthodox recursion equations. He synthesised conventional sorting algorithms by hand from a single very straightforward specification (defining sorting as selecting the ordered permutation from all the permutations of the input), which provided insights into what classes of sorting algorithms there are in addition to showing the utility of the approach.

Automatic transformation - one of his objectives has been to push automatic transformation as far as possible. In all but the very simplest examples, transformation involves the introduction of subsidiary functions. Darlington discovered that, when seeking to make a recursive definition which was not immediately possible, the partial success could in many cases be used to indicate precisely the subsidiary function required. This means-ends type reasoning he terms "forced folding", and having implemented it, is able to tackle a much wider range of problems near-automatically.

Unfree data types - his latest work is to investigate extending (at present by hand) the techniques to achieve transitions between unfree data types i.e. those data types whose operations obey certain laws.

Since this whole approach forms the basis of my work, I shall describe it in more detail:

Burstall and Darlington's set of rules manipulate recursion equations. Burstall has developed and implemented a language based around these, which he calls NPL. See the appendix for an informal introduction to NPL.

<u>Transformation Rules</u>: these act upon the recursion equations, to produce new equations. They are:

Definition:- Introduce a new recursion equation whose left hand expression is not an instance of the left hand expression of any previous equation.

Instantiation:- Introduce a substitution instance of an existing equation.

Unfolding:- If E <= E′ and F <= F′ are equations and there is some occurrence in F′ of an instance of E, replace it by the corresponding instance of E′ obtaining F′′; then add the equation F′ <= F′′.

Folding:- If E <= E′ and F <= F′ are equations and there is some occurrence in F′ of an instance of E′, replace it by the corresponding instance of E obtaining F′′; then add the equation F <= F′′. (This rule is the equivalent of Manna and Waldinger's Recursion Introduction Rule - see their section in this chapter)

Abstraction:- We may introduce a where clause, by deriving from a previous equation E <= E′ a new equation E <= E′[u1/F1,...,un/Fn] where <u1,...,un>==<F1,...,Fn>.

Laws:- We may transform an equation by using on its right hand expression any laws we have about the primitives (associativity, commutativity, etc.) obtaining a new equation.

The referenced papers contain many examples of the use of these rules to perform transformation. Within the documentation to my own system I present some of these examples (see ZAP Transformation System Primer, Chapter 4). Application of these rules preserves partial correctness of programs. Termination may be lost if folding is used without care - in practice this pitfall is easily avoided. Kott [1978] has investigated how we can restrict the use of folding so as to guarantee preserving termination.

Darlington has implemented the fold/unfold work into a system written in POP-2 on the Dec-10 at Edinburgh University. His system is designed as a research tool rather than a prototype programmer's assistant, and as such, some knowledge of the internal workings of the system is required to control it. Darlington [1977] provides an excellent account of this work.

Darlington's system uses the fold/unfold steps as the operations to manipulate definitions. The responsibilities of the user are concerned with guiding the application of these operations, as are the inbuilt heuristics of the system.

Notation: E′[u1/F1,...,un/Fn] means E′ with occurrences of F1,...,Fn replaced by u1,...,un respectively.

The <u>user's responsibilities</u> fall into several classes:

(1) He must provide an appropriate set of instantiations for the functions he wants to improve. Some of these he presents as base cases - these are unfolded completely, or until some pre-set effort bound is exceeded. The pre-set effort bound can be adjusted by the user should he expect the default setting to be inappropriate. For non base cases, the stratgey the system implements is one of carrying out a sequence of unfoldings, abstractions and applications of laws, followed by foldings.

(2) The user indicates to the system on occasions when a fold has been found whether the result is acceptable - he can veto it, accept it and stop, or accept it but request the system to search for more folds. The system itself rejects obviously undesirable folds (e.g. ones that lead to recursions which definitely do not terminate).

(3) The user supplies in advance laws which will be required for the transformation. These are in the form of equations which will be applied whenever possible during each unfolding. In Darlington's current system it is <u>not</u> possible to indicate associativity or commutativity of functions; instead explicit reduction rules tailored for the transformation being attempted must be given.

(4) The user pre-sets switches to control the search for folds. There are two such switches - ONLYTOPFOLDS and DOCLEVERFOLDS.

ONLYTOPFOLDS, if set to true, will restrict the system to seeking a fold only with the function being transformed. This is appropriate if the user is looking for a recursion involving the function in question, and does not anticipate any other folds will be required to achieve this.

DOCLEVERFOLDS, if set to true, indicates that the system may, when attempting a fold, introduce a new function in order to achieve the fold. The basis of this is a technique Darlington terms 'forced folding'. This technique, developed and implemented by Darlington, makes use of the failure to fold to indicate how to rearrange the expression being transformed to permit the fold.

e.g.

```
+++ num * num <= num
+++ num + num <= num
--- 0 + M <= M                          [1]
--- (succ N) + M <= succ (N + M)        [2]

+++ twon(num) <= num   /// twon(N) computes 2 to the power N
+++ sum(num) <= num   /// sum(N) computes 0+1+...+N
--- twon(0) <= 1                        [3]
--- twon(succ N) <= 2 * twon(N)         [4]
--- sum(0) <= 0                         [5]
--- sum(succ N) <= (succ N) + sum(N)    [6]

+++ g(num) <= num
--- g(N) <= twon(sum(N))                [7]
```

g is the function to be transformed. Consider cases 0 and succ N for its argument:

```
g(0) <= twon(sum(0))    by 7
     <= twon(0)         unfolding 5
     <= 1               unfolding 3    [8]

g(succ N) <= twon(sum(succ N))          by 7
          <= twon((succ N) + sum(N))    unfolding 6
          <= twon(succ (N + sum(N))     unfolding 2
          <= 2 * twon(N + sum(N))       unfolding 4
```

Now we are stuck - folding with the definition of g fails. Suppose we have a new function h which satisfies

```
twon(N + Y) = h(N,twon(Y))              [9]
```

then we would have

```
g(succ N) <= 2 * twon(N + sum(N))
          <= 2 * h(N,twon(sum(N)))  by 9
          <= 2 * h(N,g(N))  folding with 7      [10]
```

Thus h is just the function we require to allow a fold with g. Transforming the specification of h to get a recursive definition is relatively straightforward, we find

```
--- h(0,M) <= M
--- h(succ N,M) <= 2 * h(N,M)
```

The key to discovering h is the failure of the attempt to fold the expression 2 * twon(N + sum(N)) with the definition of g, g(N) <= twon(sum(N)).

We see that all the portions of g's definition are present within the expression - unfortunately so is the unwanted portion "N +". h is defined to move this unwanted portion outside of twon(...) so that a fold with g will become possible:

```
             twon(N + Y)      =     h(N,twon(Y))
so that  2 * twon(N + sum(N)) = 2 * h(N,twon(sum(N)))
```

The use of this technique allows the system to do some examples requiring the introduction of new functions. When the system is able to force a fold by the introduction of such a new function, the user is asked if this is acceptable, and if it is, the system is invoked recursively to transform the new function.

The technique can cause the rearrangement of expressions to permit a fold without necessarily introducing a new function.

e.g.

```
    --- g(N) <= < fib(succ N) , fib(N) >          [1]
    --- fib(succ succ N) <= fib(succ N) + fib(N)  [2]
```

(fib is the fibonnaci function)

transforming,

```
g(succ N) <= < fib(succ succ N) , fib(succ N) >
          <= < fib(succ N) + fib(N) , fib(succ N) > unfolding 2
```

attempting to fold with 1 fails, but forced folding suggests the rearrangement

```
          <= < u1 + u2 , u1>
                 where < u1 , u2 > == < fib(succ N) , fib(N) >
```

to allow a fold with 1:

```
<= < u1 + u2 , u1 > where < u1 , u2 > == g(N)
```

(5) The user sets a switch to inhibit or allow generalisation of expressions during the unfold/apply laws/fold process; if switched on, when encountering an expression containing multiple occurrences of the same variable, the generalisation is to rename these to distinct variables, and a new function, with the generalised expression as the right hand side of its defining equation, is created.

e.g. we might have

```
Cart(consset(c,X),Y) <= <: <c,b> : b in Y :> +
                         <: <a,b> : a in X , b in Y :>
```

The multiple occurrences of Y would be renamed, to give the following new function

```
newf1(c,Y1,X,Y2) <= <: <c,b> : b in Y1 :> +
                     <: <a,b> : a in X , b in Y2 :>
```

so that

```
Cart(consset(c,X),Y) <= newf1(c,Y,X,Y)
```

Again the user is asked to accept or reject the introduction of this new function, and if he accepts, the system is invoked recursively to transform it.

Most of the code to provide this generalisation facility was written by myself, and it now resides as part of Darlington's system.

Synthesis can be achieved by writing programs making use of set constructs etc., and transforming them to conventional recursive programs not making use of such constructs. Darlington has

incorporated a set of laws for these constructs which are commonly needed in transformations of this type.

The system is also able to synthesise functions defined by implicit equations - that is equations whose left hand sides are general expressions containing recursive functions among the arguments (recall that NPL expects only variables and constructors to occur there, so such definitions are unexecutable). For example, defining the inverse of REVERSE by

REVINVERSE(REVERSE(L)) <= L

the system can, from this, synthesise the definition of REVINVERSE, which turns out to be REVERSE, of course.

Typical programs whose transformation has been done using the system are:

cartesian product

fibonnacci

diagonal search

matching as inverse of substitution

a version of treesort

CHAPTER 4

USER VIEW OF SYSTEM

This chapter presents the documentation for my ZAP program transformation system. This is in two parts:

Primer

Users' Manual

The primer introduces the user to the underlying transformation method in addition to the use of the system.

The users' manual details the commands available and serves as the definitive explanation of the system.

ZAP PROGRAM TRANSFORMATION SYSTEM PRIMER

This document serves as an introduction to using the ZAP transformation system. A definitive explanation of its facilities is given in the ZAP Program Transformation System Users' Manual. ZAP is implemented in POP2 on a DEC-10.

The presentation here is in the form of three example transformations, each done first by hand, and then again as they could be tackled using the system.

The examples are

1. Scalar Product
2. Testing trees for equality of tips
3. Parsing example

The system transforms definitions written in NPL. The user is assumed to be familiar with NPL - see appendix for an informal introduction to NPL. There is no distinction between upper and lower case, but for readability I adopt the convention of using lower case for constants, constructors and functions, and upper case for variables and commands to the transformation system.

The underlying method of transformation is due to Darlington and Burstall. See Darlington [1975], [1976] and Burstall and Darlington [1977] for detailed expositions of this method, together with many examples. For an overview see Burstall and Feather [1978]. I have done the transformation of the Telegram Problem (presented in Feather [1978]) using this system.

Introduction

The ZAP transformation system is designed to aid the transformation of non-trivial programs. The system is based around the following three concepts for transformation:

A CONTEXT mechanism restricts attention of the system and the user to the relevant details for the current transformation.

The fundamental transformation step of the system involves seeking guidance from the user in the form of a GOAL consisting of a left hand side, the expression to be transformed, and a right hand side, called a PATTERN, which expresses the shape of the answer the user desires. The justification of this step is that it can be considered as the application of many steps of Burstall and Darlington's transformation method. The advantage is that it replaces many of their steps by the single step.

The system generates DEFAULT information to aid in suggesting simple goals for transformations. The user is thus able to let the system try these defaults on what he anticipates will be easy transformations, and use his insight to guide the system through more complex transformations.

Overall control of the system is achieved by giving a sequence of transformation commands. These may be typed in interactively as the transformation takes place, or stored in a disc file and called in to be used when required. In practice a convenient way to develop a transformation is to regard the sequence of commands as a program to be interactively debugged. When this "meta program" has been perfected, it, together with the initial program, serves as documentation of the final transformed program.

1 SCALAR PRODUCT

(This example is taken from Burstall and Darlington [1977])

Given a function scalar product, written ".", on vectors, defined by

$$x.y = \sum_{i=1}^{n} x_i \, y_i$$

we might wish to compute a.b + c.d

Rewriting this in NPL, we have

```
--- dot(X,Y,0) <= 0                                          [1]
--- dot(X,Y,succ N) <= dot(X,Y,N) +(X sub succ N)*(Y sub succ N)
                                                             [2]
```

(using an infix "sub" to access components of vectors)

and we want

```
--- f(A,B,C,D,N) <= dot(A,B,N) + dot(C,D,N)                  [3]
```

This is a clear definition of f, but we do not really need two separate recursive calculations (i.e. two independent loops).

The hand transformation of f goes as follows:

Consider cases 0 and succ N for f's last argument, thus

```
f(A,B,C,D,0) <= dot(A,B,0) + dot(C,D,0)     by 3
             <= 0 + 0                       unfolding 1
             <= 0                           property of 0 and +
```

```
f(A,B,C,D,succ N) <= dot(A,B,succ N) + dot(C,D,succ N)  by 3
                  <= dot(A,B,N) + (A sub succ N)*(B sub succ N) +
                     dot(C,D,N) + (C sub succ N)*(D sub succ N)
                                                 unfolding 2
                  <= dot(A,B,N) + dot(C,D,N) +
                     (A sub succ N)*(B sub succ N) +
                     (C sub succ N)*(D sub succ N)
            re-arranging using associativity and commutativity of +
                  <= f(A,B,C,D,N) + (A sub succ N)*(B sub succ N)
                     + (C sub succ N)*(D sub succ N)
                                             folding with 3
```

This completes the redefinition of f, without using dot.

The whole process has gone through the following stages:

Definitions - of f and dot.

Transforming f by considering cases - f(A,B,C,D,0) and
f(A,B,C,D,succ N).

The transformation involved -

unfolding using equations for f and dot

rearranging using properties of +

folding to get final definitions involving

f, +, *, sub, but not dot.

The transformation system commands to achieve the same process are as follows (comments for the purpose of this primer are in square parentheses):

```
START   [enters system]

DEF     [ give here NPL definitions of f and dot... ]
```

```
END

CONTEXT      [prepare to do transformation - first create the
               context in which this is to be done:]
   UNFOLD f dot    [declare that equations for f and dot are to be
                      used in unfolding process]
   USING  f [state which of the functions declared for unfolding
              we are prepared to allow in the transformed equations]
   LEMMAS ASSOCIATIVE +    [declare + to be associative]
          COMMUTATIVE +    [declare + to be commutative]
          IDENTITY + 0     [declare 0 to be identity for +. This
                             serves to reduce 0+N or N+0 to N when
                             unfolding]
   TRANSFORM       [having created context, now totally redefine f]
      GOAL f(A,B,C,D,0)         [expressions following keyword GOAL
                                  will be the left hand sides of new
      GOAL f(A,B,C,D,succ N)    equations for f]
   END  [At this point the system goes ahead trying to transform  the
```

left hand sides it has been given. f(A,B,C,D,0) expands (by unfolding and applying reductions) to 0, which, since it is a constant, is an acceptable answer. Provided expanding leads to an expression all of whose functions (if any) are constructors, constants (i.e. functions not being used for unfolding in the current context), or declared as usable (by means of the USING command), that expanded expression will be accepted as the answer. This we term a base case, since typically base-cases of recursions fall into this class.

Thus the equation f(A,B,C,D,0) <= 0 has been found.

f(A,B,C,D,succ N) unfolds to

dot(A,B,N) + (A sub succ N)*(B sub succ N) + dot(C,D,N) + (C sub succ N)*(D sub succ N)

Since this contains function dot, it is not acceptable as the answer, so the system asks the user for a "pattern". In its crudest form, a pattern is simply the right hand side we expect as the answer, which in this case would be

f(A,B,C,D,N) + (A sub succ N)*(B sub succ N)
+ (C sub succ N)*(D sub succ N)

Having typed this in, the system checks that this expression, unfolded, is equal (up to associativity and commutativity) to the unfolded left hand side. Since it is, the new equation formed by this as right hand side is added.]

DELETE f(A,B,C,D,N) [delete old definition of f]

END [to end the transform block]

STOP [to exit from the system]

This completes the commands to the system. During the transformation the system had to request the user to supply a "pattern". If the user has in mind the answer he expects/desires, he can specify this in the GOAL command by putting after the left hand side the symbol <= followed by the pattern.

From this simple example the method of transforming using the system can be seen. Instead of trying undirected unfolds, rewrites and folds, the user provides the answer he is looking for, and the system verifies this. Used in this basic way, the system is merely verifying user provided definitions. The power of the system comes into play by allowing the user to specify <u>approximately</u> the answer he

requires. The system will (if possible) fill in details to get the precise answer. Thus the user is able to manually guide the system with his intelligence and knowledge, without having to be tediously explicit.

The first way in which this is achieved is to include in a "pattern" the special function symbol $$, which the system will match to portions of the expression. It is able to match to tuple and where constructions, and functions which would be permitted in a base case - namely, constants, constructors and declared usable functions. Thus in the example we could have given as a pattern for

f(A,B,C,D,succ N) $$(A,B,C,D,N,f(A,B,C,D,N))

indicating that we expect an answer containing a call to f(A,B,C,D,N) and expressions possibly involving A,B,C,D and N, formed with usable functions constructors (e.g. succ) and constants (e.g. 0).

Using this, the transformation commands for the example are:

```
START
DEF
    [give here NPL definitions for f and dot]
END
CONTEXT
   UNFOLD f dot
   USING f
   LEMMAS ASSOCIATIVE +
          COMMUTATIVE +
          IDENTITY + 0
   TRANSFORM
      GOAL f(A,B,C,D,0)
      GOAL f(A,B,C,D,succ N) <= $$(A,B,C,D,N,f(A,B,C,D,N))
   END
   DELETE f(A,B,C,D,N)
END
STOP
```

This transformation essentially converts f to recurse on its last argument, which is of type num (natural number). Default mechanisms within the system are able to suggest straightforward recursions of this nature, and we can make use of them to both generate cases to consider, and simple patterns to try.

To indicate in the left of a goal that we are to consider cases of some argument of a function, we prefix that argument by CASESOF. The cases are derived from the right hand sides of NPL data definitions (e.g. since natural numbers are defined in NPL by DATA num <= 0 ++ succ num, for them try cases 0 and succ N).

To cause simple recursive patterns to be generated, involving recursive calls of the left side of the goal, formed by replacing some argument by its recursive case, we prefix such arguments by RECURSE. (for natural numbers, the recursive case of succ N is N). Goals containing RECURSE create simple recursive patterns consisting of $$ around all the free variables of the goal's left hand side, and that left hand side with recursive cases substituted in.

e.g. GOAL f(A,B,C,D,RECURSE succ N) produces pattern

```
$$(A,B,C,D,N,f(A,B,C,D,N))
```

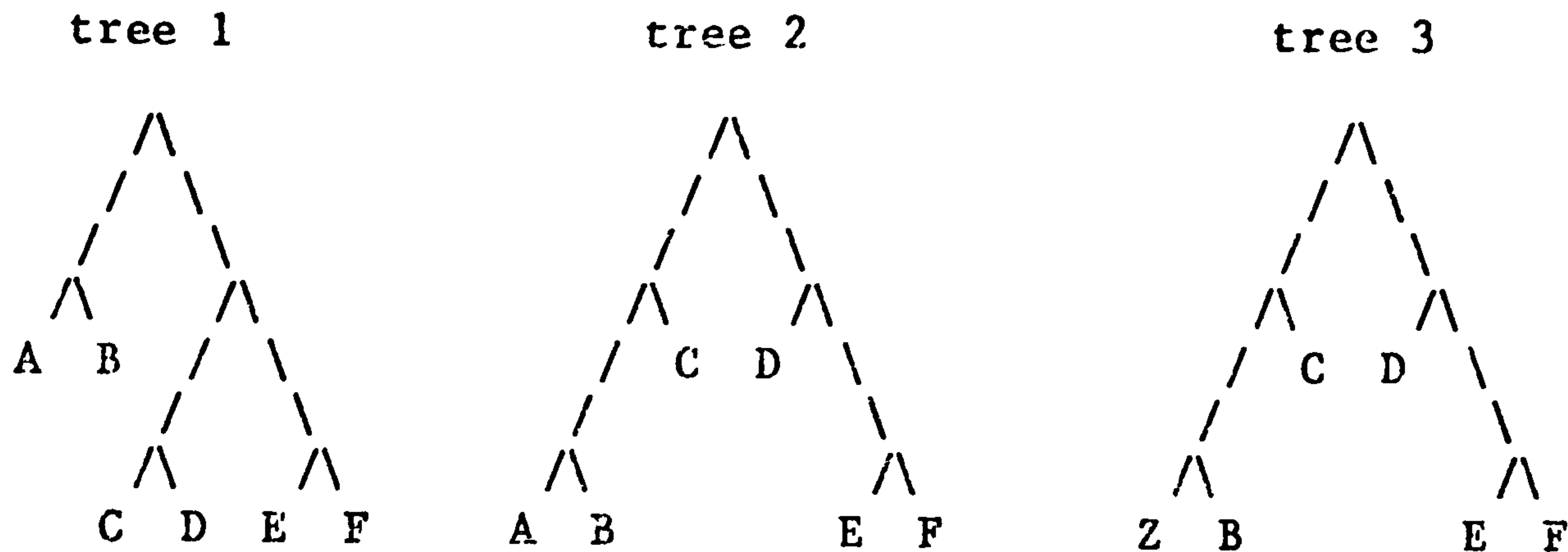
Using these features in this example, we can simplify the two goals to the following one:

```
GOAL f(A,B,C,D,RECURSE CASESOF N)
```

2 Testing trees for equality of tips

(This example is taken from Burstall and Darlington [1975])

This problem is to test whether two given binary trees have the same sequence of tips. e.g.



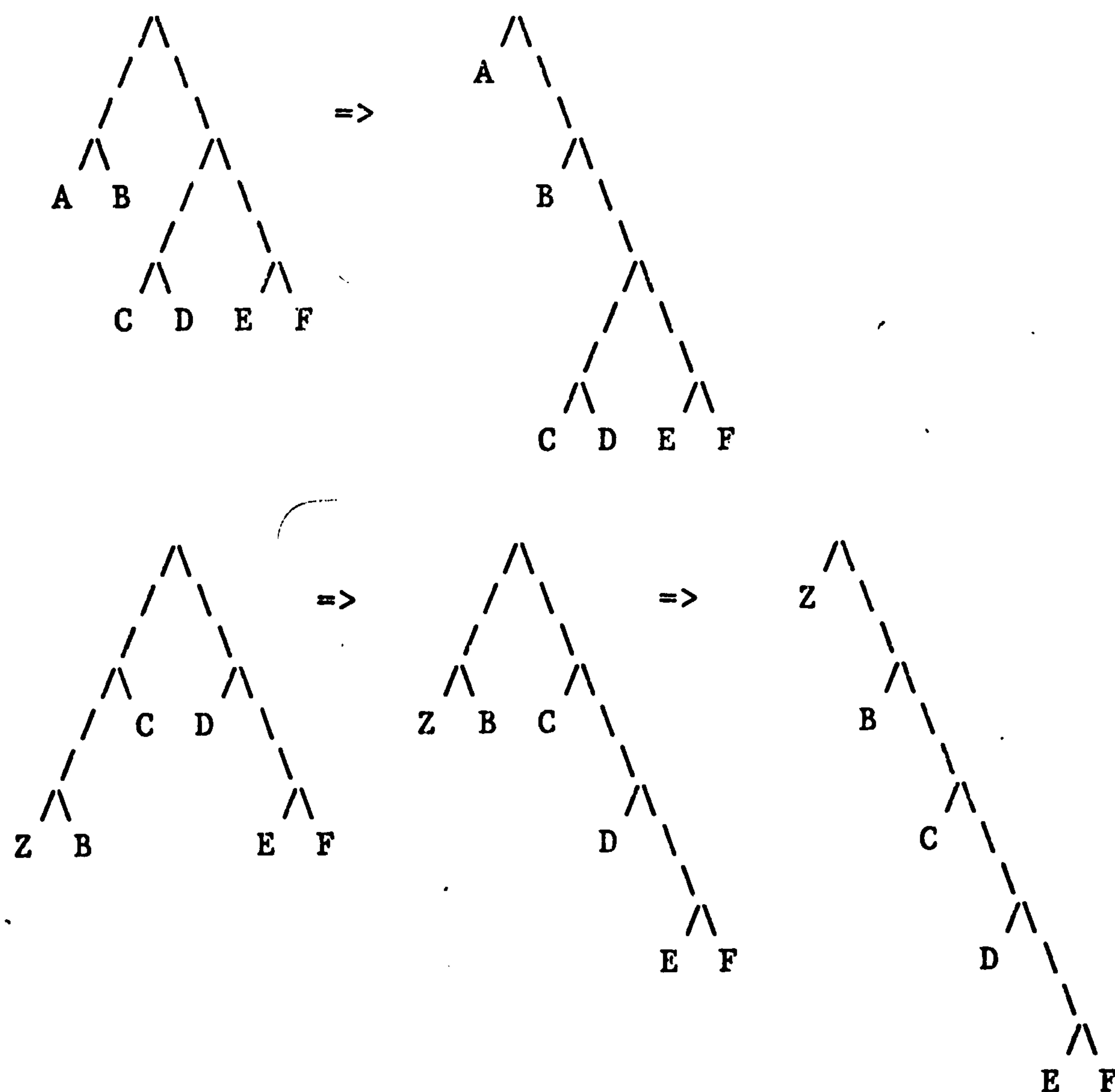
Trees 1 and 2 do have equal sequences of tips, but trees 1 and 3 do not. A straightforward solution would be to write a function to "flatten" a tree into a list of its tips, and write another function to test for equality of lists. Thus flattening trees 1 and 2 would give lists [A B C D E F] and [A B C D E F], which are equal. But this method applied to trees 1 and 3 foolishly computes the whole of the lists [A B C D E F] and [Z B C D E F] before noticing that they disagree in the very first element. We will try to obtain an improvement which avoids this.

The NPL definitions are as follows:

```
DEF
DATA trees(alfa) <= tip(alfa) ++ tree(trees(alfa),trees(alfa))

VAR A,B : alfa    VAR L1,L2 : list alfa
VAR S,T,S1,S2,T1,T2 : trees(alfa)

INF 6 <>        /// we write <> as infix append for lists
+++ list alfa <> list alfa <= list alfa
--- nil <> L2 <= L2                                        [1]
--- A::L1 <> L2 <= A::(L1 <> L2)                           [2]

+++ eqlist(list alfa,list alfa) <= list alfa
/// list equality
--- eqlist(nil,nil) <= true                                [3]
--- eqlist(nil,B::L2) <= false                             [4]
--- eqlist(A::L1,nil) <= false                             [5]
--- eqlist(A::L1,B::L2) <= A=B and eqlist(L1,L2)           [6]
```

```
+++ flatten(trees(alfa)) <= list alfa
/// list of tips of tree
--- flatten(tip(A)) <=  A::nil                                  [7]
--- flatten(tree(T1,T2)) <= flatten(T1) <> flatten(T2)          [8]

+++ eqtips(trees(alfa),trees(alfa)) <= truval
/// equality of tip sequences
--- eqtips(S,T) <= eqlist(flatten(S),flatten(T))                [9]

END
```

The improvement we seek is to compare the leftmost tip of each tree before doing unnecessary work on the remainder of the trees. One way of achieving this improvement is to restructure each tree in order to bring the leftmost tip to the top of the left branch. e.g.



Having performed this restructuring, the leftmost tips can be compared, and only if they are equal need the right branches be compared.

This neat form of restructuring is due to McCarthy. Burstall

and Darlington [1975] present an alternative way of improving the initial solution. Their improvement is similar in spirit, but uses a more general function to compare the tips of two lists of trees.

The hand transformation of eqtips to do the tree restructuring is as follows:

```
consider for each argument of eqtips the cases
   tip(A)
   tree(tip(A),T)
   tree(tree(T1,T2),T)

eqtips(tip(A),tip(B)) <= eqlist(flatten(tip(A)),tip(B))) by 9
                      <= eqlist(A::nil,B::nil) unfolding 7
                      <= A=B and eqlist(nil,nil) unfolding 6
                      <= A=B and true unfolding 3
                      <= A=B    by property of and             [10]

eqtips(tip(A),tree(tip(B),T))
          <= eqlist(flatten(tip(A)),flatten(tree(tip(B),T))) by 9
          <= eqlist(A::nil,B::flatten(T)) unfolding 7,8,2,1
          <= A=B and eqlist(nil,flatten(T)) unfolding 6
          <= A=B and false   since we know that flatten of a tree
                             contains at least one tip, so
                              eqlist(nil,flatten(T)) = false
          <= false    by property of and                       [11]

eqtips(tip(A),tree(tree(T1,T2),T)) <= false                    [12]
                             similarly to derivation of 11

eqtips(tree(tip(A),S),tree(tip(B),T))
         <= eqlist(flatten(tree(tip(A),S)),flatten(tree(tip(B),T)))
         <= A=B and eqlist(flatten(S),flatten(T))
                    unfolding 8,7,2,1,6
         <= A=B and eqtips(S,T) folding with 9                 [13]

eqtips(tree(tip(A),S),tree(tree(T1,T2),T))
         <= eqlist(flatten(tree(tip(A),S))
                  ,flatten(tree(tree(T1,T2),T))) by 9
         <= eqlist(flatten(tree(tip(A),S)),
                  (flatten(T1)<>flatten(T2))<>flatten(T) )
                               unfolding 8
         <= eqlist(flatten(tree(tip(A),S)),
                  flatten(T1)<>(flatten(T2)<>flatten(T)) )
                       by associativity of <>
         <= eqlist(flatten(tree(tip(A),S)),
                  flatten(tree(T1,tree(T2,T))))
                       folding with 8
         <= eqtips(tree(tip(A),S),tree(T1,tree(T2,T)))         [14]
                       folding with 9
```

similarly, we find

```
eqtips(tree(tip(A),S),tip(B)) <= false                              [15]

eqtips(tree(tree(S1,S2),S),tip(B)) <= false                         [16]

eqtips(tree(tree(S1,S2),S),tree(tip(B),T))
          <= eqtips(tree(S1,tree(S2,S)),tree(tip(B),T))             [17]

eqtips(tree(tree(S1,S2),S),tree(tree(T1,T2),T))
      <= eqtips(tree(S1,tree(S2,S)),tree(T1,tree(T2,T)))            [18]
```

Equations 10 - 18 form the new defintion of eqtips. In order to perform the same transformations using the system, we would give the following commands:

```
CONTEXT
   USING eqtips and
   UNFOLDALL eqtips
   LEMMAS ASSOCIATIVE <>
          COMMUTATIVE eqlist
          --- eqlist(nil,flatten(T)) <= false
          --- eqlist(A::nil,L1<>L2)
                    <= (eqlist(A::nil,L1) and eqlist(nil,L2)) or
                       (eqlist(A::nil,L2) and eqlist(nil,L1))
          --- eqlist(nil,L1<>L2) <= eqlist(nil,L1) and eqlist(nil,L2)
   TRANSFORM
      GOAL eqtips(tip(A),tip(B))
      GOAL eqtips(tip(A),tree(tip(B),T))
      GOAL eqtips(tip(A),tree(tree(T1,T2),T))
      GOAL eqtips(tree(tip(A),S),tip(B))
      GOAL eqtips(tree(tip(A),S),tree(tip(B),T))
                                         <= $$(A,B,eqtips(S,T))
      GOAL eqtips(tree(tip(A),S),tree(tree(T1,T2),T))
                    <= $$(eqtips(tree(tip(A),S),tree(T1,tree(T2,T)))
      GOAL eqtips(tree(tree(S1,S2),S),tip(B))
      GOAL eqtips(tree(tree(S1,S2),S),tree(tip(B),T))
                      <= $$(eqtips(tree(S1,tree(S2,S)),tree(tip(B),T)))
      GOAL eqtips(tree(tree(S1,S2),S),tree(tree(T1,T2),T))
                 <= $$(eqtips(tree(S1,tree(S2,S)),tree(T1,tree(T2,T)))
   END
   DELETE eqtips(S,T)
END
```

The goal mechanism has saved us the many small steps of unfolding, applying properties of functions, and folding. Even so, it is still tedious to have to give nine goals corresponding to each combination of the cases the two arguments of eqtips can take.

Equally tedious is the need to specify rather simple patterns for several of these goals.

The key to our solution is to consider cases
tip(A), tree(tip(A),T), tree(tree(T1,T2),T)
for arguments of type trees(alfa).
When transforming a call involving the case tree(tip(A),T) look for a recursion involving a call on T. Similarly, when a call involves the case tree(tree(T1,T2),T) look for a recursion involving a call to tree(T1,tree(T2,T)).

The system can do much of the work for us if we first give <u>type</u> information about type trees(alfa). The form this takes is

```
TYPEINFO T <= tip(A)
           <= tree(tip(A),T) , T
           <= tree(tree(T1,T2),T) , tree(T1,tree(T2,T))
```

T is a variable of type trees(alfa), and following each "<=" is one of the cases we want to split this type into, each such case followed by its recursive case(s) if any.

Once this type information has been given, the single goal

GOAL eqtips(RECURSE CASESOF S, RECURSE CASESOF T)

will be expanded into the nine goals, with simple recursive patterns generated for each one. Thus the transformation commands to make the improvement need only be:

```
CONTEXT
  TYPEINFO T <= tip(A)
             <= tree(tip(A),T) , T
             <= tree(tree(T1,T2),T) , tree(T1,tree(T2,T))
  USING eqtips and
  UNFOLDALL eqtips
  LEMMAS ASSOCIATIVE <>
         COMMUTATIVE eqlist
         --- eqlist(nil,flatten(T)) <= false
         --- eqlist(A::nil,L1<>L2)
                  <= (eqlist(A::nil,L1) and eqlist(nil,L2)) or
                     (eqlist(A::nil,L2) and eqlist(nil,L1))
         --- eqlist(nil,L1<>L2) <= eqlist(nil,L1) and eqlist(nil,L2)
```

```
   TRANSFORM
      GOAL eqtips(RECURSE CASESOF S, RECURSE CASESOF T)
   END
   DELETE eqtips(S,T)
END
```

The system has a default mechanism to generate simple type information if we do not provide any, however for the type trees(alfa) it would generate the equivalent of

```
   TYPEINFO T <= tip(A)
              <= tree(S,T) , S,T
```

hence for the solution we are aiming for here, we must supply our own more sophisticated type information.

3 Parsing Example

(This example is taken from Darlington [1976])

So far the transformations have involved restructuring the definitions in terms of already defined functions. This example will illustrate how the system can assist in deriving auxiliary functions when these are required to permit the desired restructuring.

The problem is to take input in the form of a stream of text (letters and spaces) on fixed length records and convert this stream into a stream of words, one word per record.

For example, the sentence THE CAT SAT would be recorded, using records of length 3 as

```
[[THE][ CA][T S][AT]]
```

and the required output is

[[THE][CAT][SAT]]

The naive program works by first flattening the input structure. Thus the above input would be converted into

[THE CAT SAT]

This is then restructured into the desired output. We can optimise this two pass program into a one pass one. Lists are used to represent the records and words.

The NPL definitions are as follows:

```
DEF
DATA character <= ap ++ sp
/// ap's are letters and sp is the space

+++ translate(list list character) <= list list character
--- translate(CLL) <= parse(flatten(CLL))                [1]

+++ flatten(list list character) <= list character
--- flatten(nil) <= nil                                  [2]
--- flatten((C::CL)::CLL) <= C::flatten(CL::CLL)         [3]
--- flatten(nil::CLL) <= flatten(CLL)                    [4]

+++ parse(list character) <= list list character
--- parse(CL) <= firstw(CL) :: parse(restw(CL))          [5]

+++ firstw(list character) <= list character
/// selects the first word off a flat list
--- firstw(nil) <= nil                                   [6]
--- firstw(ap::CL) <= ap::firstw(CL)                     [7]
--- firstw(sp::CL) <= nil                                [8]

+++ restw(list character) <= list character
/// removes the first word and following spaces
--- restw(nil) <= nil                                    [9]
--- restw(ap::CL) <= restw(CL)                           [10]
--- restw(sp::CL) <= skipsp(CL)                          [11]

+++ skipsp(list character) <= list character
/// removes all spaces up to next letter
--- skipsp(nil) <= nil                                   [12]
--- skipsp(sp::CL) <= skipsp(CL)                         [13]
--- skipsp(ap::CL) <= ap::CL                             [14]
```

The hand translation of this goes as follows:

First, define a new function, parsel :

```
+++ parsel(list list character) <= tuple2(list character,
                                          list list character)
```

Parsel is to take the original structured list and return a pair consisting of the first word of the list, and the parse of the rest of the list, thus:

```
--- parsel(CLL) <= <firstw(flatten(CLL)),parse(restw(flatten(CLL))>
```

Now transform this definition of parsel by considering cases nil, nil::CLL, (ap::CL)::CLL and (sp::CL)::CLL for its argument:

```
parsel(nil) <= <firstw(flatten(nil)),parse(restw(flatten(nil))>
            <= <nil,nil>

parsel(nil::CLL) <= <firstw(flatten(CLL)),parse(restw(flatten(CLL))>
                                     unfolding
                 <= parsel(CLL)       folding

parsel((ap::CL)::CLL) <= <ap::firstw(flatten(CL::CLL)),
                           parse(restw(flatten(CL::CLL))>
                                     unfolding
                      <= <ap::W,WL> where <W,WL> ==
                          <firstw(flatten(CL::CLL)),
                           parse(restw(flatten(CL::CLL)))
                                     abstracting
                       <= <ap::W,WL> where <W,WL> == parsel(CL::CLL)
                                     folding

parsel((sp::CL)::CLL) <= <nil,parse(skipsp(flatten(CL::CLL)))>
                                     unfolding
```

We now come to a non trivial part of the transformation: We would like to define parsel((sp::CL)::CLL) recursively, but the unfolded expression is not quite of the appropriate form to allow us to do this. If we introduce an auxiliary function skipspz such that skipsp(flatten(CLL)) = flatten(skipspz(CLL)) then we can get a recursive definition. skipspz does on the structured list what skipsp does on the flattened list. Darlington has developed a method of deriving the definition of auxiliary functions when seeking to

force a fold with some function (see Darlington [1975] for details of this). For the present, let us assume that skipspz has been introduced, and proceed from there:

```
                    <= <nil,parse(skipsp(flatten(CL::CLL)))>
                              (repeating previous line)
                    <= <nil,parse(flatten(skipspz(CL::CLL)))>
                              using skipspz
                    <= <nil,firstw(flatten(skipspz(CL::CLL)))::
                           parse(restw(flatten(skipspz(CL::CLL))))>
                              unfolding
                    <= <nil,W::WL> where <W,WL> ==
                         <firstw(flatten(skipspz(CL::CLL))),
                          parse(restw(flatten(skipspz(CL::CLL))))>
                              abstracting
                    <= <nil,W::WL> where <W,WL> ==
                                         parsel(skipspz(CL::CLL))
                              folding
```

Now define translate using parsel:

```
translate(CLL) <= firstw(flatten(CLL))::parse(restw(flatten(CLL)))
                              unfolding
            <= W::WL where <W,WL> ==
                    <firstw(flatten(CLL)),parse(restw(flatten(CLL)))>
                               abstracting
            <= W::WL where <W,WL> == parsel(CLL)
```

Finally we have to synthesise skipspz. Recall that skipspz must satisfy

```
          flatten(skipspz(CLL)) = skipsp(flatten(CLL))
```

Thus we need

```
flatten(skipspz(nil)) = skipsp(flatten(nil))
                      = nil
                      = flatten(nil)   folding with [2]
for which we need:
skipspz(nil) <= nil

flatten(skipspz(nil::CLL)) = skipsp(flatten(nil::CLL))
                           = skipsp(flatten(CLL))
                           = flatten(skipspz(CLL))
                     using our defining equation for skipspz
for which we need
skipspz(nil::CLL) <= skipspz(CLL)

flatten(skipspz((ap::CL)::CLL) = skipsp(flatten((ap::CL::CLL))
                               = skipsp(ap::flatten(CL::CLL))
                               = ap::flatten(CL::CLL)
                               = flatten((ap::CL)::CLL) folding [3]
for which we need
skipspz((ap::CL)::CLL) <= (ap::CL)::CLL
```

```
flatten(skipspz((sp::CL)::CLL)) = skipsp(flatten((sp::CL)::CLL))
                                = skipsp(sp::flatten(CL::CLL))
                                = skipsp(flatten(CL::CLL))
                                = flatten(skipspz(CL::CLL))
                          using our defining equation for skipspz
for which we need
skipspz((sp::CL)::CLL) <= skipspz(CL::CLL)
```

This completes the hand transformation.

In the hand transformation, function parsel was tailored specifically for translate. Hence it would be appropriate to combine its introduction with the redefinition of translate. This would have the advantage of ensuring that the appropriate definition of a new function for use by translate (as well as its type declaration) would be created as we transformed translate.

Essentially we wish to express translate(CLL) as

W::WL where <W,WL> == parsel(CLL)

parsel being the new function.

The way we would do this in the system is to prefix the name of the new function by the special symbol && within our goal, thus:

GOAL translate(CLL) <= W::WL where <W,WL> == &&parsel(CLL)

The process works by making the new function act as a function variable to match a portion of the expression, in much the same way that $$ does, but in this case the instantiation becomes the definition of the new function rather than a portion of the right hand side of the transformed equation.

It is here that we see a need for another class of usable functions. We intend introducing a new function who's definition involves functions parse, flatten, restw and firstw. If we simply declare all these functions to be usable, the expansion of

translate(CLL) will turn out to be a base case. We require a class of functions which may occur in our answer, but which may not occur in a base case. Because we intend this new class of usable functions to be more restricted in their application, we term them usable, but RESTRICTED.

They are declared in the USING command, after an extra keyword - RESTRICTED.

e.g. USING RESTRICTED flatten firstword

Such usable restricted functions may occur in the right hand side of a transformed equation only if we explicitly give them in the pattern, and/or match them to new functions (by means of the && facility). They inhibit acceptance of the expression as a base case, and are NOT matched by $$. Intuitively, this is the class of functions that we expect to occur in the answer, but who's use we wish to exert some control over. The example done with the aid of this feature will illustrate these points. The commands given to the system are as follows:

```
START
DEF
    [give here NPL definitions of translate, flatten, parse, firstw,
     restw and skipsp]
END

CONTEXT
   UNFOLD translate parse
   USING RESTRICTED flatten firstw restw parse    [these are usable,
                                                   but restricted]
   TRANSFORM
      GOAL translate(CLL) <= W::WL where <W,WL> == &&parsel(CLL)
   END
END
```

The system finds as definition of parsel:

parsel(CLL) <= <firstw(flatten(CLL)),parse(restw(flatten(CLL)))>

which it adds to the NPL equations, having made the type definition for parsel. Thus we have redefined translate and introduced parsel all in a single step. Now redefine parsel :

```
CONTEXT
   UNFOLD parsel firstw flatten parse restw
   USING parsel RESTRICTED skipsp flatten
   TRANSFORM
      GOAL parsel(nil)           [we expect this to be a base case]
      GOAL parsel(nil::CLL) <= $$(parsel(CLL))
      GOAL parsel((ap::CL)::CLL) <= $$(ap,parsel(CL::CLL))
      GOAL parsel((sp::CL)::CLL) <= $$(parsel(&&skipspz(CL::CLL)))
   END
   DELETE parsel(CLL)
END
```

parsel(nil) expands to simply <nil,nil> i.e. a base case.

parsel(nil::CLL) (i.e. at end of current section of input) and parsel((ap::CL)::CLL) (i.e. found alphanumeric of current word) recurse simply, to give:

parsel(nil::CLL) <= parsel(CLL)

parsel((ap::CL)::CLL) <= <ap::W,WL>

where <W,WL> == parsel(CL::CLL)

parsel((sp::CL)::CLL) has no immediate recursion with parsel(CL::CLL). It corresponds to reaching a space in the input, terminating the word we were building up. We expect, therefore, that we should skip spaces in the input until an alphanumeric (or end of input) is reached, and continue with parsel from that point. However, skipsp acts on the flattened input, list character, whereas parse works on list list character, so we need a new skipspz which will be analogous to skipsp. The way we introduce this is by using

the && facility in the pattern. The system finds

```
parsel((sp::CL)::CLL) <= <nil,W::WL>
                    where <W,WL> == parsel(skipspz(CL::CLL))
```

and as a definition of skipspz,

```
flatten(skipspz(CLL)) <= skipsp(flatten(CLL))
```

Darlington terms such a definition an "implicit" definition, since it is not in the usual form of having the defined function on the outside of the left hand side. The system always attempts to rearrange such a definition to bring the function being defined to the outside of the left hand side. To do this it will introduce inverses, thus:

```
skipspz(CLL) <= iflatten(skipsp(flatten(CLL)))

iflatten(flatten(CLL)) <= CLL
```

iflatten is the inverse of flatten.

In general, inverses will not be uniquely defined, if they exist at all. The intention is to transform a definition making use of inverses into one without any such uses, applying only inverse properties of the introduced inverse functions.

In this manner we use the system to transform skipspz:

```
CONTEXT
   UNFOLD skipsp flatten skipspz iflatten
   USING skipspz
   LEMMAS --- iflatten(C::flatten(CL::CLL)) <= (C::CL)::CLL
   TRANSFORM
      GOAL skipspz(nil)
      GOAL skipspz(nil::CLL) <= $$(skipspz(CLL))
      GOAL skipspz((ap::CL)::CLL)
      GOAL skipspz((sp::CL)::CLL) <= $$(skipspz(CL::CLL))
   END
   DELETE skipspz(CLL)
END
```

This is another opportunity to make use of type information to simplify our goals:

Firstly, introduce our own information for type list list char

```
        TYPEINFO CLL <= nil
                     <= nil::CLL , CLL
                     <= (ap::CL)::CLL , CL::CLL
                     <= (sp::CL)::CLL , CL::CLL
```

once this has been given, our 4 goals can be encapsulated in one by:

```
          GOAL skipspz(RECURSE CASESOF CLL)
```

The system finds

```
skipspz(nil) <= nil

skipspz(nil::CLL) <= skipspz(CLL)

skipspz((ap::CL)::CLL) <= (ap::CL)::CLL

skipspz((sp::CL)::CLL) <= skipspz(CL::CLL)
```

ZAP PROGRAM TRANSFORMATION SYSTEM USERS' MANUAL

INTRODUCTION

This manual describes an interactive system to allow a user to transform NPL programs into NPL programs. NPL is a first-order recursion equation language. See the appendix for an informal introduction to NPL. The underlying method of transforming is due to Darlington and Burstall, see Darlington [1975], [1976] and Burstall and Darlington [1977] for detailed expositions of this method, with many examples. For introductory examples to informally demonstrate the use of the system, see the ZAP Program Transformation System Primer (previous section). The NPL interpreter and parser were written by Rod Burstall, and my system makes use of code written by John Darlington which links with NPL.

The syntax of the control language of the system is given in B.N.F. Underlined words, called "reserved words", are represented by the same words without underlining in an actual program. There is no distinction between upper and lower case. For readability I adopt the convention of lower case for constants, constructors and functions, and upper case for reserved words and variables.

Description of the system is in the following stages:

1. Control of system
2. Transformation features
3. Syntax of control language

1. CONTROL OF SYSTEM

To run the system, at monitor level type the MIC command

/ZAP [450,463]

When initialisation is complete, 'READY' will be printed. The user is then inside POP-2, with the transformation system compiled. The system transforms from and to NPL. This manual assumes the user is familiar with NPL. The NPL interpreter is available and may be used to test initial and final programs. Any NPL definitions (subject to restrictions detailed in [1.2]) made outside the transformation system are passed in upon entry to the system.

1.1 Overall Control

The syntax of the overall control of the system is as follows:

```
<control> ::= START <control contents> STOP

<control contents> ::= <empty> |
                       <control command> <control contents>

<control command> ::= <def block> | <introduce block> |
                      <context block> | <infile command> |
                      <typeinfo command> | <val block> |
                      <state command> | </// command> |
                      <delete comand> | <write block> |
                      <showspecs block>
```

Thus to enter the system, type START. After this the system is then waiting for one of the following:

DEF block - to provide the initial NPL program to be transformed [1.2]

INTRODUCE block - to introduce auxiliary definitions [1.3]

CONTEXT block - to transform definitions [1.4]

INFILE command	- to cause commands to be taken from a disc file [1.5]
TYPEINFO command	- to provide information about data types [1.6]
VAL block	- to evaluate an expression [1.7]
STATE command	- to save current state of definitions or restore a previously saved state [1.8]
/// command	- to include comments. Reads up to next reserved word, ignoring all intermediate POP2 items [1.9]
DELETE command	- to delete equations [1.10]
WRITE block	- to write definitions out to a disc file [1.11]
SHOWSPECS block	- to display the definition of a function in terms of the original functions [1.12]
STOP	- to exit from the system

These are now described in detail.

1.2 The DEF block

Syntax : <def block> ::= DEF <NPL> END

<NPL> is any sequence of NPL definitions that may appear in a "DEF...END" block of normal NPL, with the following restrictions:

The type overloading of functions (i.e. using the same function symbol to indicate different operations depending upon the types of its arguments) is not permitted.

The reserved words of the transformation system may not be used - these are ASSOCIATIVE COMMUTATIVE CONTEXT DELETE DISPLAY GOAL IDENTITY INFILE INTRODUCE LEMMAS RESTORE RESTRICTED SAVE START STOP

TRANSFORM TYPEINFO UNFOLD UNFOLDALL USING WRITE WRITEALL.

In addition, the following symbols have special meaning to the system: $$ && AUTO CASESOF RECURSE

NPL if/ifnot clauses may be used in the DEF and INTRODUCE blocks, but will be converted to make use of the conditional function, COND. Where the syntax demands an NPLEXPN, if/ifnot clauses may not be used.

The purpose of this command is to present the initial program, which serves as the specification.

1.3 The INTRODUCE block

Syntax : <introduce block> ::= INTRODUCE <NPL> END

This acts precisely as the DEF block. Its purpose is to introduce auxiliary definitions which are not to be regarded as part of the specification.

1.4 The CONTEXT block

The purpose of this block is to set up a context in which transformations take place. Amongst the features declared within a context are equations to be used for unfolding, and lemmas to aid in unfolding and matching.

To understand the uses of all the features, we must first understand the way in which transformation is performed. Transformations change the right hand sides of equations and may

introduce new functions and their definitions. The fundamental way in which a transformation is carried out is as follows:

A function with instantiated arguments is supplied. This, if the transformation succeeds, will be the left hand side (henceforth referred to as L.H.S.) of a new equation. Also supplied is an expression which is to be the right hand side (henceforth referred to as R.H.S.) of the new equation. It is valid to introduce such a new equation if we can show that the L.H.S., unfolded using existing equations and rewrite rules, is equal to the R.H.S., similarly unfolded. Schematically,

```
 L.H.S.                            R.H.S.
   |                                  |
   |                                  |
   V unfold                    unfold V
   |                                  |
   |                                  |
expanded                         expanded
 L.H.S.            =              R.H.S.
```

If they are equal, we add the equation L.H.S. <= R.H.S. to the existing NPL definitions.

The context in which a transformation occurs is set up by the commands of the CONTEXT block. These are now described, and further details of the transformation process are given in [1.4.5]

Syntax :

```
<context block> ::= CONTEXT <context contents> END

<context contents> ::= <empty> |
                       <context command> <context contents>

<context command> ::= <using command> | <unfold command> |
                      <lemmas command> | <typeinfo comand> |
                      <introduce block> | <infile command> |
                      <transform block> | <delete command> |
                      <val block> | </// command> |
                      <state command> | <context block> |
```

<showspecs block>

Briefly, these act as follows:

UNFOLD command - declares functions whose equations will be used to unfold L.H.S. and R.H.S. [1.4.1]

USING command - declares which functions may occur in the R.H.S. of a transformed equation. [1.4.2]

LEMMAS command - introduces rewrite rules for unfolding, and indicates functions to be associative and/or commutative. [1.4.3]

TRANSFORM block - transform equations of a function [1.4.4]

CONTEXT block - nested entry to CONTEXT block. Within nested CONTEXT blocks, an inner block inherits the context set up by the surrounding block.

The remaining commands perform the same functions as they did at the outer level of the system.

1.4.1 The UNFOLD Command

Syntax : <unfold command> ::= UNFOLD <namelist> |
UNFOLDALL <namelist>

This command names functions whose equations are to be used to unfold the L.H.S. and R.H.S. in transformations. Successive UNFOLD commands in the same CONTEXT block supplement the list of such functions. (on entry to the system, this list is empty).

The basic form of this command is to give the names of functions to be used after the reserved word UNFOLD. However, it is often convenient to use all the relevant NPL equations for unfolding, without having to name them explicitly. By giving function names after the reserved word UNFOLDALL, all the equations which are used directly or indirectly by those named functions will be included for unfolding.

e.g. UNFOLD length +

states that equations for length and + are to be used in unfolding.

e.g. UNFOLDALL length

states that all equations used directly or indirectly by length are to be used in unfolding.

1.4.2 The USING command

Syntax : <using comand> ::= USING <using contents>

<using contents> ::= <namelist> |
<namelist> RESTRICTED <namelist>

This command specifies functions additional to constants and constructors (constants are recognised by having no equations), which may occur in the R.H.S. of a transformation. There are two classes of such functions - restricted and unrestricted. Only functions named after the reserved word RESTRICTED are restricted. The use of the distinction between restricted and unrestricted is described later. See [1.4.5] and [2.1].

Successive USING commands in the same CONTEXT block supplement each of these classes, the last mention of a function name determining which class it is in. Initially only COND (conditional function) and = (equality function) are usable (both unrestricted).

e.g. USING length succ

declares length and succ to be usable, unrestricted.

e.g. USING + RESTRICTED append length

declares + to be usable, unrestricted, and append and length to be usable, restricted.

1.4.3 The LEMMAS command

Syntax :

```
<lemmas command> ::= LEMMAS <lemmas contents>

<lemmas contents> ::= <empty> |
                      <npleqnlist> <lemmas contents> |
                      ASSOCIATIVE <namelist> <lemmas contents> |
                      COMMUTATIVE <namelist> <lemmas contents> |
                      IDENTITY <NAME> <NPLEXPN> <lemmas contents>

<npleqnlist> ::= <empty> | --- <npleqn> <npleqnlist>

<npleqn> ::= <NPLEXPN> <= <NPLEXPN>
```

This command does one of four things:

Firstly, re-write rules can be provided, which will be used in the unfolding of the L.H.S. and R.H.S. Since these rewrite rules will be applied whenever possible during the unfolding, no rule or set of rules should be capable of being repeatedly applied indefinitely. (e.g. including the rules A*(B+C) <= (A*B)+(A*C) and

(A*B)+(A*C) <= A*(B+C) would lead to infinite looping.)

Secondly, functions may be declared to be associative. The effect is to cause the equality test between expanded L.H.S. and expanded R.H.S. during transformation to take the associativity into account - i.e. it will test for equality up to associativity of these functions.

Thirdly, functions may be declared to be commutative, so that the equality test will take this into account. Further, such a declaration causes symmetric versions of equations and reductions for the commutative function to be added if not already present, so as to enhance the unfolding process. e.g.
if we have equations for eqlist, a function to test equality between two lists:

```
--- eqlist(nil,A2::L2) <= false
--- eqlist(A1::L1,A2::L2) <= A1=A2 and eqlist(L1,L2)
```

and reduction

```
--- eqlist(nil,L1<>L2) <= eqlist(nil,L1) and eqlist(nil,L2)
```

Then declaring eqlist to be commutative will add the equation

```
--- eqlist(A2::L2,nil) <= false
```

and the reduction

```
--- eqlist(L1<>L2,nil) <= eqlist(nil,L1) and eqlist(nil,L2)
```

to be used during unfolding.

Lastly, the identity for a (binary) function can be declared. After reserved word IDENTITY, the function name is given, followed by the expression which is its identity. This will cause applications of that function to its identity to be reduced during the unfolding

of R.H.S.'s and L.H.S.'s.

Since these are user-provided details, the validity of the transformation will depend upon the validity of these details. Successive LEMMAS commands in the same block supplement these details.

e.g. LEMMAS --- N + succ M <= succ (N + M)
ASSOCIATIVE + - *
COMMUTATIVE + - *
IDENTITY + 0

These add rewrite rule N + succ M <= succ (N + M), declare functions +, - and * to be both associative and commutative(adding symmetrical versions of all equations and reductions for these functions) and declare 0 to be the identity of +.

1.4.4 The TRANSFORM Block

Syntax :

<transform block> ::= TRANSFORM <goal list> END

<goal list> ::= <empty> | <goal> <goal list>

<goal> ::= GOAL <left hand side> |
GOAL <left hand side> <= <pattern>

<left hand side> ::= <NPLEXPN>

<pattern> ::= <NPLEXPN>

These are the commands which cause transformations to be carried out.

After the initial reserved word, a list of goals is given. Each goal is an expression to become the left hand side of a new equation, optionally followed by "<=" and a pattern. A pattern is the means by which the desired R.H.S. of the new equation is given. In its simplest form, this will be the exact expression the user expects as the R.H.S.

Used in this fashion, the transformation system acts as a verifier, merely checking user-provided definitions. Patterns can also be used to aid in the discovery of new definitions - achieved by giving as a pattern only the approximate shape of the answer desired, letting the system fill in the details. This is explained fully in section 2.

If a left hand side is mentioned more than once in the goals, then the corresponding patterns (if any) will be tried in the order in which they appeared.

If all the patterns for a left hand side fail, the system will enter into an interactive dialogue with the user to get another pattern, or be told to abandon the attempted transformation.

Provided all the left hand sides are successfully transformed, the system will then take the following action:

If any of the new equations have left hand sides identical to those of existing equations in NPL, the existing equations are removed from NPL and saved by adding them to the "specifications list". Saving them enables us to later look back and see how functions were originally defined. The SHOWSPECS command, [1.12], does this.

All the new equations are added to NPL.

The context in which the transformation takes place will be that at the point where the TRANSFORM block occurs.

e.g. Suppose we have funnyplus, so called because it behaves like plus but for the order of its arguments in its recursive call, defined as follows:

```
+++ funnyplus(num,num) <= num
--- funnyplus(0,M) <= 0
--- funnyplus(succ N,M) <= succ funnyplus(M,N)
```

Having created the appropriate context, we might say

```
TRANSFORM
   GOAL funnyplus(succ N,M) <= succ funnyplus(N,M)
END
```

This will attempt to transform the second of the equations for funnyplus. If the goal given is successful, this will change to

```
--- funnyplus(succ N,M) <= succ funnyplus(N,M)
```

and save the old equation by adding it to the "specifications list".

e.g. Suppose we have length, append and lap defined as follows:

```
+++ length(list atom) <= num
+++ append(list atom,list atom) <= list atom
+++ lap(list atom,list atom) <= num
--- length(nil) <= 0
--- length(A::AL) <= succ length(AL)
--- append(nil,AL1) <= AL1
--- append(A::AL,AL1) <= A::append(AL,AL1)
--- lap(AL,AL1) <= length(append(AL,AL1))
```

then, having created a suitable context, we could say

```
TRANSFORM
   GOAL lap(nil,AL1) <= length(AL1)
   GOAL lap(A::AL,AL1) <= succ   lap(AL,AL1)
END
```

This will attempt to redefine lap, and if the goal is successful, will give

```
--- lap(nil,AL1) <= length(AL1)
--- lap(A::AL,AL1) <= succ lap(AL,AL1)
```

1.4.5 Further details about transformation

Commands for preparing the context for a transformation have now been described. Two important details of the transformation process can now be mentioned:

(i) There are three classes of functions;

Unusable

Usable, unrestricted

Usable, restricted

If all the functions within the expanded L.H.S. are either usable, unrestricted, or unusable but simply constructors or constants (within the current context), and the expanded L.H.S. contains no iterative expressions, this is termed a "basecase". The equation

```
L.H.S. <= expanded L.H.S.
```

is added to the equations, and no R.H.S. in the goal is required. Typically base cases of recursive functions fall into this class, hence the terminology "basecase".

(ii) The only unusable functions that an R.H.S. of a goal may contain are constructors and constants (within the current context).

1.5 The INFILE command

Syntax : <infile command> ::= INFILE <FILENAME>

By default, the input of commands to the system is taken from the user's terminal. Commands stored in a disc file may be used instead. To cause command input to come from such a file, use the INFILE command.

e.g. INFILE [FILE1.EXT]

Would cause command input to come from file FILE1.EXT

Upon end-of-file being reached, input reverts to the user's terminal.

1.6 The TYPEINFO command

Syntax : <typeinfo command> ::= TYPEINFO <nplexpn> <typeinfo cases>

<typeinfo cases> ::= <= <nplexpnlist> |
<= <nplexpnlist> <typeinfo cases>

This command is one of the features present to reduce the effort of using the system. The user can save himself some later effort by specifying some information about data types in advance.

Type information will be used for two purposes - suggesting cases of an argument to consider, and aiding in the generation of simple patterns.

Splitting transformation into cases is done by looking at the cases an argument can take. The type of the argument will influence the cases we would consider.

e.g. For an argument of type num (natural number), we might consider the cases

0 and succ N

alternatively, we might consider

0 , 1 and succ succ N

e.g. For an argument of type trees(alfa) (binary trees of alfa's), we might consider the cases

tip(A) and tree(T1,T2)

alternatively, we might consider

tip(A) , tree(tip(A),T) and tree(tree(T1,T2),T)

Generating a simple pattern will involve forming simple recursive calls to the function being transformed. The type of the function's arguments determine the recursive calls which could be made.

e.g. When transforming a call to a function with argument of type num, in the form succ N; f(succ N,...), we might generate a pattern including the recursive call with argument N; f(N,...) for call f(succ succ N,...) the pattern might include calls f(N,...) and/or f(succ N,...)

e.g. When transforming a call to a function with argument of type trees(alfa), in the form tree(T1,T2); g(tree(T1,T2),...) we might generate a pattern including the recursive calls with arguments T1 and T2; g(T1,...) and g(T2,...)

The essence of the above is that for each data type there may be some way (or ways) of splitting the type into cases, and for some of these cases there may be "recursive" cases which can be used to form

recursive calls in patterns.

e.g.

type	case	recursive cases (if any)
num	0	
	succ N	N
trees(alfa)	tip(A)	
	tree(T1,T2)	T1 , T2
trees(alfa)	tip(A)	
	tree(tip(A),T)	T
	tree(tree(T1,T2),T)	tree(T1,tree(T2,T))

The TYPEINFO command is a means of specifying such information. The syntax for presenting the information is very similar to the above layout. Instead of naming the type, any expression of that type suffices. Each cases is preceeded by the symbol "<=", and followed by its recursive cases (if any), separated by commas.

e.g.

```
TYPEINFO N <= 0
           <= succ N , N

TYPEINFO T <= tip(A)
           <= tree(T1,T2) , T1 , T2

TYPEINFO T <= tip(A)
           <= tree(tip(A),T) , T
           <= tree(tree(T1,T2),T) , tree(T1,tree(T2,T))
```

If the user has not provided type information about a type, the system applies a default mechanism to generate such information if it is required. This works by looking at the NPL DATA declaration for the type in question; each of the cases in the DATA declaration becomes a case in the type information. Recursive cases are formed by spotting occurrences of the type within the cases of the DATA declaration.

e.g. DATA num <= 0 ++ succ num

generates

```
TYPEINFO N <= 0
           <= succ N , N
```

e.g. DATA list(alfa) <= nil ++ alfa :: list(alfa)

generates

```
TYPEINFO L <= nil
           <= A::L , L
```

For straightforward problems, the default type information will often suffice. The user can override this by providing his own type information when the default is not appropriate (as in the eqtips example presented in the Primer).

Using type information

In order to cause a transformation goal to be expanded into several goals by considering cases of some argument, prefix the argument by CASESOF. The current type information will be used to suggest the cases.

e.g. GOAL funnyplus(CASESOF N , M)

expands to (using default type information for type num)

GOAL funnyplus(0 , M)

GOAL funnyplus(succ N , M)

Prefixing several arguments by CASESOF will create all combinations of cases

e.g. GOAL funnyplus(CASESOF N , CASESOF M)

expands to

GOAL funnyplus(0 , 0)

GOAL funnyplus(0 , succ M)

<u>GOAL</u> funnyplus(succ N , 0)

<u>GOAL</u> funnyplus(succ N , succ M)

The details of how to generate simple recursive patterns are left until section [2.2]

1.7 The VAL block

Syntax : <val block> ::= <u>VAL</u> <NPLEXPN> <u>END</u>

The NPL expression is evaluated using the current NPL equations. Hence this can be used to test and compare functions.

1.8 The STATE command

Syntax : <state command> ::= <u>SAVE</u> | <u>RESTORE</u> <NUMBER>

The current state of NPL definitions and the specification list may be saved by giving the command SAVE. The system responds with the number of the state where they have been saved, which may be restored later by giving command RESTORE, followed by the appropriate number. Successive SAVE commands cause the state to be saved in successively numbered locations.

1.9 The /// command

Syntax : </// command> ::= /// <pop2 itemlist>

<pop2 itemlist> ::= <empty> | <POP2 ITEM> <pop2 itemlist>

This provides a comment facility. After ///, pop2 items are read and discarded until a reserved word is encountered. The reserved words are ASSOCIATIVE COMMUTATIVE CONTEXT DELETE DISPLAY GOAL IDENTITY INFILE INTRODUCE LEMMAS RESTORE RESTRICTED SAVE START STOP TRANSFORM TYPEINFO UNFOLD UNFOLDALL USING WRITE. Since a pop2 string is a single pop2 item, reserved words may occur within such comments provided they are within string quotes.

e.g.

/// this is a comment

e.g.

/// 'this is a comment within string quotes@

1.10 The DELETE command

Syntax : <delete command> ::= DELETE <nplexpnlist>

This command removes equations from NPL. After the reserved word DELETE, give the left hand sides of the equations to be removed. Equations deleted from NPL are saved on the "specifications list", which may be accessed later to determine how functions were originally defined. The SHOWSPECS command does this accessing, see [1.12] for details of this.

4.11 The WRITE block

Syntax : <write block> ::= WRITE <FILENAME> <namelist> END | WRITEALL <FILENAME> <namelist> END

This command allows current NPL definitions to be written to a disc file.

If the reserved word WRITE is given, only the equations of the named functions will be written to file.

If the reserved word WRITEALL is used, all equations of both the named functions and any function used directly or indirectly by them will be written to file.

The destination file is specified after the appropriate reserved word, and followed by the function names.

Output is an NPL DEF...END block, including the type declarations for all the functions whose equations are to be listed, variable declarations for all variables used, and finally the equations themselves. After this, the specifications of all used functions (in terms of the original functions) are printed.

e.g.

WRITE [LEN.NPL] alength

might write the following:

```
DEF
+++ alength(list atom,num) <= num
VAR N : num
VAR A : atom
VAR AL : list atom
--- alength(nil,N) <= N
--- alength(A::AL,N) <= alength(AL,succ N)
END
```

SPECIFICATIONS
alength(AL,N) <= N + length(AL)

1.12 The SHOWSPECS block

Syntax : <showspecs block> ::= SHOWSPECS <namelist> END

This command displays the definitions of the named functions in terms of the original functions.

e.g. if function alength had originally been defined by

alength(AL,N) <= N + length(AL)

and later transformed to get a recursive definition,

SHOWSPECS alength END

would cause the original definition to be displayed.

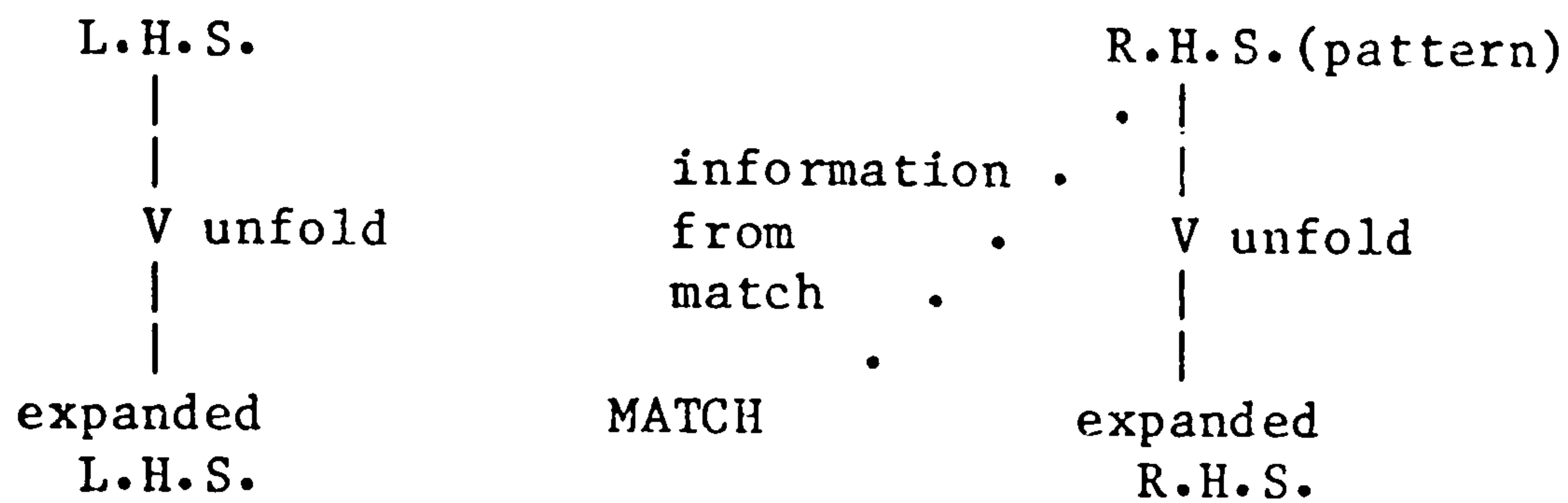
2. TRANSFORMATION FEATURES

The basic transformation system has now been presented. Some features have been mentioned but not fully explained. It is certainly possible to use the system as described, but the following facilities will ease the use of the system.

2.1 Patterns for transformations

As stated earlier, the approach to transformation is to supply an L.H.S. and R.H.S. which are expanded and compared. There are often occasions when the user has an idea of the form of R.H.S. he desires, and he may wish to specify that form without having to give every detail.

The key to making this possible is the inclusion in the R.H.S. of function variables which MATCH to functions - hence the terminology "pattern" for the supplied R.H.S. The transformation process then becomes:



Bindings formed in the match between expanded L.H.S. and expanded R.H.S. are used to instantiate the R.H.S. to form the answer.

There are two types of function variables, used for different purposes.

The simpler use of them is to match to portions of the expression and thus save the user the need to specify those portions precisely in his pattern. Function variables to do this form of matching are represented by the symbol $$.

The other use of them is to match to portions of the expression which are to be taken as the definition of a new function, the portions within the expression being replaced by calls to the new function. Function variables to do this form of matching are represented by the symbol &&, followed by a (new) name to serve as the new function's name.

2.1.1 $$ Matching

The symbol $$ within a pattern is used for a function variable to match to portions of the expression. $$ will match to constructors (including where and <...> constructs), usable-unrestricted functions, and functions which are constants within the current context (i.e. have no equations declared for use in unfolding), provided they have not been declared restricted.

Note, therefore, that $$ will NOT match to iterative expressions, any restricted functions, or unusable functions with equations in the current context.

e.g. with the following definitions

```
+++ sum(list num) <= num
+++ squares(list num) <= list num
--- sum(nil) <= 0
--- sum(N::L) <= N + sum(L)
--- squares(nil) <= nil
--- squares(N::L) <= (N*N) :: squares(L)
+++ sumsquares(list num) <= num
--- sumsquares(L) <= sum(squares(L))
```

we might perform the following transformation

```
CONTEXT
   UNFOLD sumsquares sum squares
   USING sumsquares
   TRANSFORM
      GOAL sumsquares(nil)
      GOAL sumsquares(N::L) <= $$(N,sumsquares(L))
   END
END
```

The first GOAL has only the L.H.S. sumsquares(nil). This unfolds to 0, which, being a constructor, is a basecase, so the equation

sumsquares(nil) <= 0

is added.

The second GOAL has L.H.S. sumsquares(N::L). This unfolds to

(N*N) + sum(squares(L)), not a basecase since functions sum and squares occur within it, neither of which have been declared as usable. The R.H.S. is the pattern $$(N,sumsquares(L)), which unfolds to $$(N,sum(squares(L))). Matching the expanded pattern and expanded L.H.S. succeeds, binding $$ to

lambda x y . x*x + y

(note that although neither * nor + have been declared usable, they do not have equations within this context, hence may occur within the binding of $$). The $$ in the unexpanded R.H.S. is instantiated to give the answer

sumsquares(N::L) <= N*N + sumsquares(L)

Clearly the above example was so simple that we could have easily anticipated the precise form of the answer and typed it as the goal without recourse to using $$. However, the advantages of matching become apparent both in tackling less trivial examples, and in using patterns generated by the system itself - section [2.2].

If the pattern contains several occurrences of the symbol $$, these need not bind to the same expression, so the single symbol $$ suffices for multiple occurrences of function variables in patterns.

2.1.2 && Matching

The purpose of including the $$ function variable within a pattern was to save having to specify detailed portions of answers. An alternate way in which matching can be of help is to introduce new functions whose definitions arise from extracting portions of expressions being transformed.

Function variables for this purpose are represented by the

special symbol && followed by a new name which, if the match is successful, will be taken as the name of the newly defined function.

Function variables of this nature may match to any constructors, usable functions (both restricted and unrestricted), and functions which are constants within the current context (i.e. have no equations defined for unfolding). Thus they fail to match only with unusable functions with equations for unfolding in the current context.

Ambiguity is avoided by not permitting any function variable (of either type) to occur within an argument of a function variable of the && type.

e.g. with the following definitions

```
+++ wordsof(list char) <= list word
--- wordsof(ap(A)::CL) <= firstword(ap(A)::CL) ::
                          wordsof(ap(A)::CL - firstword(ap(A)::CL))
```

we might perform the following transformation

```
CONTEXT
   UNFOLD wordsof firstword
   USING wordsof RESTRICTED firstword -
   TRANSFORM
      GOAL wordsof(ap(A)::CL) <= W::wordsof(REMCL)
             where <W,REMCL> == &&wordandrem(ap(A)::CL)
   END
END
```

For this single goal, the L.H.S. is wordsof(ap(A)::CL). This unfolds to

firstword(ap(A)::CL) :: wordsof(ap(A)::CL - firstword(ap(A)::CL))

which is not a base case, because it contains resticted functions - and firstword.

The R.H.S. is W::wordsof(REMCL)

where <W,REMCL> == &&wordandrem(ap(A)::CL)

this does not unfold any further.

Matching the expanded L.H.S. and expanded R.H.S. binds &&wordandrem to

```
lambda x . <firstword(x) , x - firstword(x)>
```

Instead of using this to instantiate the variable in the answer, this becomes the definition of new function wordandrem:

```
--- wordandrem(x) <= <firstword(x) , x - firstword(x)>
```

The transformed answer includes a call to this new function:

```
--- wordsof(ap(A)::CL) <= W :: wordsof(REMCL)
                  where <W,REMCL> == wordandrem(ap(A)::CL)
```

The system generates the type declaration for the new function, and within the context declares the function to be usable, restricted, so that further goals in the same context (if any) may make use of it.

From the above example the usefulness of the restricted class of usable functions can be seen - by declaring functions to be restricted, the only way they may occur in the answer is to be explicitly mentioned in the pattern, or to be incorporated as part of the definition of a new function. We therefore have stricter control over occurrences of such functions in the answer.

2.2 Supplementing and generating patterns

In section [1.6] the Typeinfo command was described. The means by which the user, or the system itself, could specify information about a data type were explained. This information included a list of 'recursive cases' for each case we split the data type into. This information can be of use when seeking a suitable R.H.S. during a

transformation. Very often we expect the transformed equation for some function to involve recursive calls to that function.

e.g. when transforming function lap,

+++ lap(list atom,list atom) <= num

we may expect the equation for lap(A::AL,AL1) to involve the recursive call lap(AL,AL1)

i.e. lap is recursing on its first argument

The recursive calls arise by replacing argument(s) in the L.H.S. by their recursive cases. Some data types may have more than one recursive case (e.g. a binary tree would have its right and left branches) which would give rise to more than one recursive call. In order to make the specification of such patterns easier, the following facilities have been provided:

When giving a goal, to indicate we wish a pattern to recurse upon an argument of the left hand side, prefix that argument by the symbol RECURSE.

e.g. GOAL lap(RECURSE A::AL , AL1) <= ....

causes the pattern

$$(A,AL,AL1,lap(AL,AL1))

to be generated

The generated pattern consists of $$ with argumments all free variables of the L.H.S., plus recursive calls formed by substituting recursive cases for the indicated arguments within the L.H.S. Occurrences of the special constant AUTO within the pattern specified at the right hand side of the goal will then be replaced by this derived pattern.

e.g.

GOAL lap(RECURSE A::AL,AL1) <= AUTO

is equivalent to

```
    GOAL lap(A::AL,AL1) <= $$(A,AL,AL1,lap(AL,AL1))
```

e.g.

```
    GOAL funnyplus(RECURSE succ N,M) <= AUTO + AUTO
  is equivalent to
    GOAL funnyplus(succ N,M) <= $$(N,M,funnyplus(N,M)) +
                                $$(N,M,funnyplus(N,M))
```

If several arguments are prefixed by RECURSE, this causes several recursive calls to be included.

e.g.

```
    GOAL funnyplus(RECURSE succ N,RECURSE succ M)
  is equivalent to
    GOAL funnyplus(succ N,succ M) <=
             $$(N,M,funnyplus(N,succ M),funnyplus(succ N,M),
                                     funnyplus(N,M))
```

Prefixing an argument which does not have a recursive case (e.g. a variable) by RECURSE will not cause any extra recursive call within the derived pattern.

There are some defaults to simplify further:

If the L.H.S. of a goal contains occurrences of RECURSE, and no "<=" followed by a pattern has been provided, AUTO is assumed as the pattern. If the pattern provided contains occurrences of AUTO, but there are no occurrences of RECURSE within the L.H.S., each argument of the L.H.S. is prefixed by RECURSE.

e.g.

```
    GOAL lap(RECURSE A::AL,AL1)
  is equivalent to
    GOAL lap(RECURSE A::AL,AL1) <= AUTO
```

e.g.

```
    GOAL lap(A::AL,AL1) <= AUTO
  is equivalent to
    GOAL lap(RECURSE A::AL,RECURSE AL1) <= AUTO
```

Goals making use of both this facility and CASESOF have the CASESOF arguments expanded before RECURSE is applied.

e.g.

```
    GOAL lap(RECURSE CASESOF AL,AL1)
  expands to
    GOAL lap(nil,AL1) <= $$(AL1)
    GOAL lap(A::AL,AL1) <= $$(A,AL,AL1,lap(AL,AL1))
```

3. SYNTAX OF CONTROL LANGUAGE

```
<control> ::= START <control contents> STOP

<control contents> ::= <empty> | <control command> <control contents>

<control command> ::= <def block> | <introduce block> |
                      <context block> | <infile command> |
                      <typeinfo command> | <val block> |
                      <state command> | <///command> |
                      <delete command> | <write block> |
                      <showspecs block>

<def block> := DEF <NPL> END

<introduce block> ::= INTRODUCE <NPL> END

<context block> ::= CONTEXT <context contents> END

<context contents> ::= <empty> |
                       <context command> <context contents>

<context command> ::= <using command> | <unfold command> |
                      <lemmas command> | <typeinfo command> |
                      <transform block> | <delete command> |
                      <introduce block> | <infile command> |
                      <val block> | <///command> |
                      <state command> | <context block> |
                      <showspecs block>

<infile command> ::= INFILE <FILENAME>

<val block> ::= VAL <NPLEXPN> END

<///command> ::= /// <pop2 itemlist>

<state command> ::= SAVE | RESTORE <NUMBER>

<delete command> ::= DELETE <nplexpnlist>
```

```
<write block> ::= WRITE <FILENAME> <namelist> END |
                  WRITEALL <FILENAME> <namelist> END

<showspecs block> ::= SHOWSPECS <namelist> END

<using command> ::= USING <using contents>

<using contents> ::= <namelist> |
                     <namelist> RESTRICTED <namelist>

<unfold command> ::= UNFOLD <namelist> | UNFOLDALL <namelist>

<lemmas command> ::= LEMMAS <lemmas contents>

<lemmas contents> ::= <empty> |
                      <npleqnlist> <lemmas contents> |
                      ASSOCIATIVE <namelist> <lemmas contents> |
                      COMMUTATIVE <namelist> <lemmas contents> |
                      IDENTITY <NAME> <NPLEXPN> <lemmas contents>

<typeinfo command> ::= TYPEINFO <nplexpn> <typeinfo cases>

<typeinfo cases> ::= <=  <nplexpnlist> |
                     <=  <nplexpnlist> <typeinfo cases>

<transform block> ::= TRANSFORM <goal list> END

<goal list> ::= <empty> | <goal> <goal list>

<goal> ::= GOAL <left hand side> |
           GOAL <left hand side> <=  <pattern>

<pattern> ::= <NPLEXPN>

<left hand side> ::= <NPLEXPN>

<namelist> ::= <empty> |
               <NAME> <namelist>

<nplexpnlist> ::= <NPLEXPN> | <NPLEXPN> , <nplexpnlist>

<npleqnlist> ::= <empty> | --- <npleqn> <npleqnlist>

<npleqn> ::= <NPLEXPN> <=  <NPLEXPN>

<pop2 itemlist> ::= <empty> | <POP2 ITEM> <pop2 itemlist>
```

The following are assumed:

<NUMBER> is any unsigned integer

<NAME> is any identifier suitable for use as the name of an NPL function

<NPL> is any sequence of NPL definitions which would be permitted within an NPL DEF...END block, subject to the restrictions

outlined in [1.2]

<NPLEXPN> is any expression not using if/ifnot constructs.

Note: the following reserved words may not occur in NPL definitions or expressions to be used by the system:
ASSOCIATIVE COMMUTATIVE CONTEXT DELETE DISPLAY GOAL IDENTITY INFILE INTRODUCE LEMMAS RESTORE RESTRICTED SAVE START STOP TRANSFORM TYPEINFO UNFOLD UNFOLDALL USING WRITE WRITEALL

In addition, the following symbols have special meanings to the system: $$ && AUTO CASESOF RECURSE

<FILENAME> is any file specification suitable for pop2

Index to main syntactic elements in User's Manual

<context block> [1.4]
<control> [1.1]
<def block> [1.2]
<delete command> [1.10]
<goal> [1.4.4]
<introduce block> [1.3]
<infile command> [1.5]
<left hand side> [1.4.4]
<lemmas command> [1.4.3]
<pattern> [1.4.4] and [2.1]
<showspecs block> [1.12]
<state command> [1.8]
<transform block> [1.4.4]
<typeinfo command> [1.6]
<unfold command> [1.4.1]
<using command> [1.4.2]
<val block> [1.7]
<write block> [1.11]
</// command> [1.9]

CHAPTER 5

TRANSFORMING LARGE EXAMPLES

In this chapter I consider the transformation of large programs using my system. Firstly, I discuss commonly used tactics to introduce beneficial changes into the structure of a program, and the need for a strategy to guide the use of these on non-trivial problems. Then I present two large examples which I have transformed using my system.

5.1 TRANSFORMATION TACTICS

The lowest level operations underlying the transformations my system performs are the rules of unfolding and folding . The system raises the user above this bottom level - for individual transformations he provides patterns which express the structure of the answer required, and the system tries to fill in the details, maintaining equivalence with the existing definition by linking with a series of small unfold/fold etc. steps.

The structure of the initial program will typically not be at all geared towards efficiency. Each transformation causes some change in structure, and the intention is to improve the efficiency of the initial program dramatically by a series of changes. The efficiency improvements come about from one of several classes of

manipulations: My investigations have been in the domain of transformations between recursion equation programs, where major changes in structure are carried out. As a final stage to my transformations, I introduce extra arguments to functions. These arguments act as accumulators for results being built up, so as to get the recursion equations into a form suitable for straightforward conversion into an imperative language. The term "accumulators" originates from the work of Moore [1974].

The bulk of my attention has been on the transformations between recursive programs. These transformations are used to make improvements in two classes, each of which sugests a 'tactic' to be used in transformation to improve efficiency. The tactics are:

5.1.1 Combining Tactic

This tactic is intended to overcome the inefficiencies resulting from the embedding of two or more functions. When we have an expression of the form f(g(x)) we seek to improve this by defining a new function --- fg(x) <= f(g(x)) and seeking a recursive definition of fg which will compute the result in one pass rather than two.

e.g. suppose we have function squares, which given a list of numbers, returns the list of those numbers squared:

```
--- squares(nil) <= nil
--- squares(N::L) <= N*N :: squares(L)
```

and function sum to sum the numbers of a list

```
--- sum(nil) <= 0
--- sum(N::L) <= N+sum(L)
```

then we can compute the sum of the squares of a list of numbers

by

```
sum(squares(L))
```

This is obviously inefficient, as an intermediate structure, the list of squared numbers, is produced completely and then consumed. The combining tactic suggests we define

```
--- sqsum(L) <= sum(squares(L))
```

and find a recursive definition, which turns out to be

```
--- sqsum(nil) <= 0
--- sqsum(N::L) <= N*N + sqsum(L)
```

In simple cases, the improvement from applying this tactic can also be achieved by evaluating the equations using lazy evaluation (call by need) which causes evaluation to be delayed until absolutely necessary. In more complex problems, the attempt to produce a recursive definition of the newly introduced function may not be straightforward, and lazy evaluation not sufficient to achieve the improvements that transformations can provide.

5.1.2 Tupling Tactic

This tactic is intended to overcome the inefficiencies resulting from there being two or more separate calls to differing functions with the same argument. When an expression is of the form ...f(x)...g(x)... we seek to improve this by defining a new function: fandg(x) <= < f(x),g(x) > which computes the pair of results simultaneously.

e.g. suppose we wish to compute the standard deviation of a list of numbers; we will require both the sum of those numbers, and the

sum of their squares. These can be profitably computed together by defining

```
--- sumandsqsum(L) <= < sum(L),sqsum(L) >
```

and transforming this to get

```
--- sumandsqsum(nil) <= < 0 , 0 >
--- sumandsqsum(N::L) <= < N+N1 , N*N+N2 >
                                   where <N1,N2> == sumandsqsum(L)
```

This will save us the computation involved in traversing each argument separately. Work by Pettorossi [1977] has demonstrated that this may also allow improvements in memory utilization. In non-trivial tuplings, some of the common calculations of each previously separate function need not be done more than once.

5.2 TRANSFORMATION STRATEGIES

On sizeable problems, there will be many ways of applying the transformation tactics to achieve improvements. It might be the case that order of application of tactics is not crucial, and whichever we choose, we will end up with a suitably efficient program. If, however, we do not follow any systematic approach, we increase the risk of getting lost in a morass of detail during the transformation. In particular, we would find it hard to draw parallels from already completed transformations. If we modify the protoprogram and need to adjust the transformation to accomodate the modification, a strategy behind the transformation would help us pinpoint the portions potentially needing change.

A strategy is required to guide the application of the tactics. In the examples that follow, I use a straighforward strategy which

appears at least to satisfy our requirements of standardising the approach, and leading to a suitable final program.

The strategy I adopt is the very basic one of always seeking improvements from the 'inside out' By this I mean that if some function f uses another function g, first improve g before tackling f. When there are several embedded functions to combine, unless we feel confident enough to try them all at once, combine them from the inner outwards e.g. f(g(h(X))) - first combine gh(X) <= g(h(x)), then combine fgh(X) <= f(gh(X)).

I do not claim that this basic strategy is the "best" in any sense. There may be a more appropriate strategy, or there may even be no general strategy which guides us through the transformations by the easiest path. I argue only that this is systematic, and works in practice.

5.3 THE TELEGRAM PROBLEM

This example originates from Henderson and Snowdon [1972]. They used it as a programming exercise, developed a solution, and then discovered that their solution behaved unsatisfactorily at the less explicitly specified boundary of the problem. I am not suggesting some program as the definitive solution to the telegram problem. Since the specification in English is somewhat ambiguous, I would argue that some degree of choice is left to the programmer. My concern is that it should not be necessary to analyse an efficient program to determine what choices have been made.

5.3.1 English Specification

"A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer area where it is to be processed. The words in the telegrams are separated by sequences of blanks and each telegram is delimited by the word "ZZZZ". The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words. The words "ZZZZ" and "STOP" are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word."

Since then it has been considered by Ledgard [1974], Zahn [1976], McKeeman [1976], Jones [1976], Jackson [1977] and Schwarz [1977].

e.g.

input

(available in sections of length 15):

```
7  KLINGON SHIP
S APPROACHING S
TOP  SEND REINF
ORCEMENTS STOP
ZZZZ    TOO LATE
  ZZZZ ZZZZ
```

output

```
telegram: 7 KLINGON SHIPS APPROACHING STOP SEND
          REINFORCEMENTS STOP
** overlength word present **
count of chargeable words: 6


telegram: TOO LATE
count of chargeable words: 2
```

For consideration here I do not take the entire telegram problem, but go only as far as producing a list of telegrams with their corresponding statistics, omitting the final step of neatly listing these. I feel that this last step does not introduce any new difficulties, and only enlarges the problem.

The transformational approach suggests that first we design a very simple solution to the problem - the protoprogram - and then transform this to achieve efficiency. The protoprogram serves as the precise specification of our solution resolving the ambiguities in the English specification; its behaviour will be easy to determine, and to modify if desired.

5.3.2 Design Of Protoprogram

Firstly a definition of the data types to represent input and output:

```
data alphanumeric <= cha ++ chb ++ ... ++ chz ++ ch0 ++ ... ++ ch9
data char <= ap(alphanumeric) ++ sp
data instream <= in(list list char)
data message <= me(telegram,statistics)
```

Four types have been defined:

alphanumeric represents non-blank characters in the input

char represents all characters, including spaces (sp)

instream represents the input, stated to be made available through a buffer area. Thus in(nil) is end-of-input, in(nil::CLL) is an empty buffer and remaining stream CLL, and in((C::CL)::CLL) is a buffer containing first character C, remaining characters CL, and remaining stream CLL.

list message is to be our output. Each message contains a telegram and its statistics (data types which we will define later).

The overall task will be performed by function getmessages: pictorially,

```
                          ------------         getmessages          ----------------
                          | instream |------------->------------| list message |
                          ------------                              ----------------
```

+++ getmessages(instream) <= list message

Now we break this down into smaller stages. We need to convert the instream into a list of telegrams, and from this list compute the list of messages.

```
------------ gettels ----------------- messagesof ----------------
| instream |---->----| list telegram |----->------| list message |
------------          -----------------             ----------------
```

+++ gettels(instream) <= list telegram

+++ messagesof(list telegram) <= list message

Since a message can be produced from its telegram independent of the other messages, let this be done by a smaller function, messof:

```
------------ messof -----------
| telegram |--->----| message |
------------          -----------
```

+++ messof(telegram) <= message

gettels needs further decomposition. A telegram is a list of words, where words are sequences of non-blanks to be found in the input separated by blank(s).

```
data telegram <= te(list word)
data word <= wo(list alphanumeric)
```

Since the buffer boundaries have no significance, we might as well "flatten" the input into a simple list of characters before breaking this into a list of words, and these into a list of telegrams.

```
----------flatten------------wordsof------------telsof---------------
|instream|--->---|list char|--->---|list word|--->--|list telegram|
----------       ------------       ------------      ---------------
```

```
+++ flatten(instream) <= list char
+++ wordsof(list char) <= list word
+++ telsof(list word) <= list telegram
```

flatten we expect to have no difficulty with.

wordsof must compute a list of words from a list of characters. This would be easy if we had a smaller function (firstword) to produce just the first word, for then we could use this to get the first word, and compute the remaining list of characters by simply subtracting the characters of that first word.

```
------------- firstword --------      -------- wtocl -------------
| list char |------>------| word |      | word |--->---| list char |
-------------             --------      --------       -------------

+++ firstword(list char) <= word       +++ wtocl(word) <= list char
```

telsof must compute a list of telegrams from a list of words. This is similar to wordsof, and we will find it useful to have a function (firsttel) to compute just the first telegram from a list of words.

```
------------- firsttel ------------  ------------ ttowl -------------
| list word |---->-----| telegram |  | telegram |--->---| list word |
-------------          ------------  ------------       -------------
```

```
+++ firsttel(list word) <= telegram

                                  +++ ttowl(telegram) <= list word
```

messof is the function that given a telegram, produces a message. In addition to the telegram itself, the message contains statistics, which for this problem are a count of chargeable words, and a boolean which will be true if and only if the telegram contains no overlength words. We introduce a function statsof to compute statistics:

```
  data statistics <= st(num,truval)


------------ statsof --------------
| telegram |---->----| statistics |
------------          --------------

  +++ statsof(telegram) <= statistics
```

This will use two smaller functions:

charge to count chargeable words

```
 ------------- charge -------
 | list word |---->---| num |      +++ charge(telegram) <= num
 -------------        -------
```

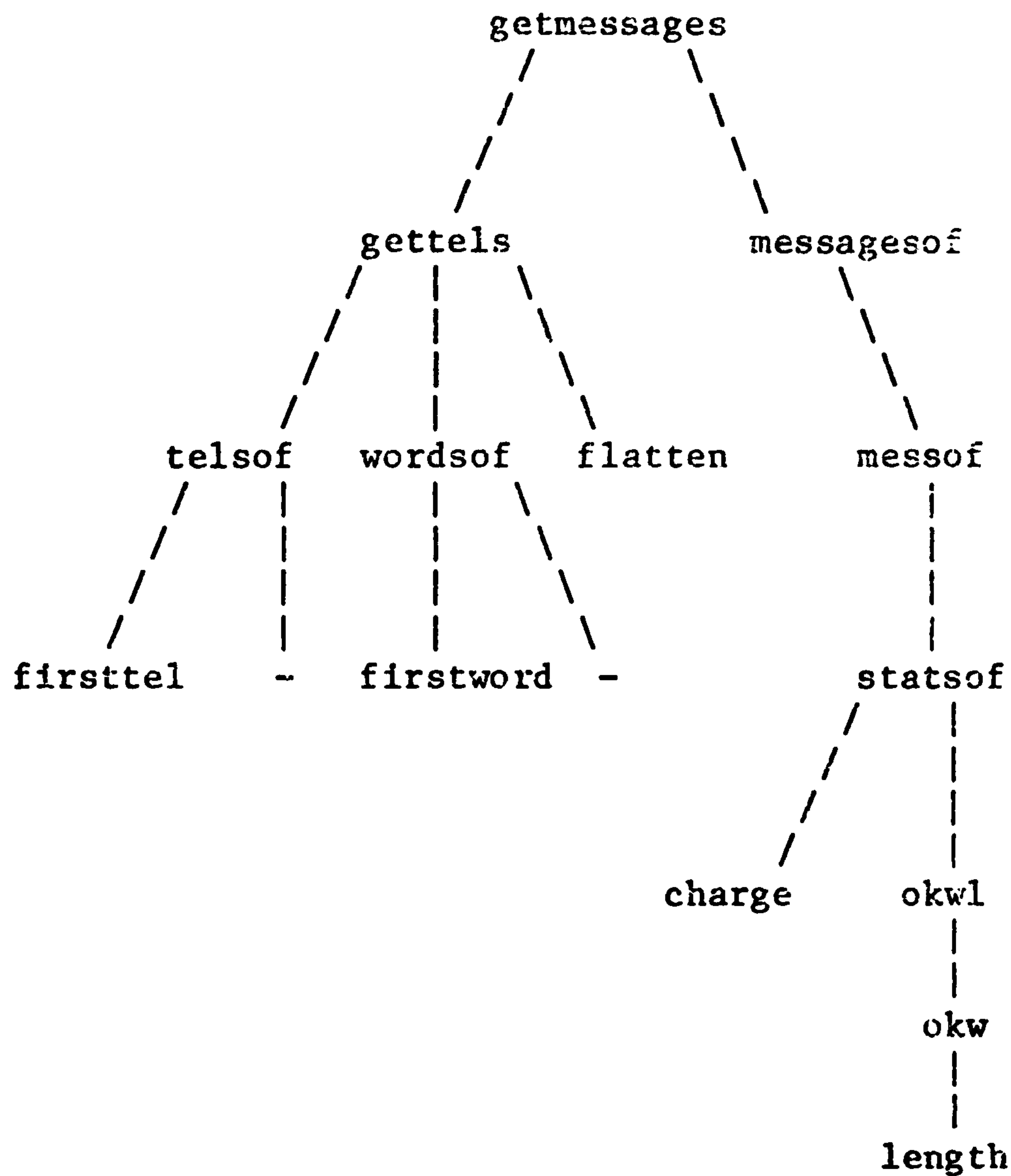
and okwl to check that no words are overlength

```
 ------------- okwl ----------
 | list word |--->--| truval |      +++ okwl(list word) <= truval
 -------------      ----------
```

which itself uses a function to check a single word

```
        -------- okw ----------
        | word |-->--| truval |      +++ okw(word) <= truval
        --------     ----------
```

The overall structure of the protoprogram has been designed. The following diagram shows pictorially the structure of the protoprogram (in these structure diagrams a line joining two functions indicates that the higher function calls the lower one).



Now the details remain to be filled in - a relatively straightforward task. It is here that the precise behaviour of the solution will be determined.

5.3.3 NPL Protoprogram

```
DEF

DATA alphanumeric <= cha ++ chb ++ chc ++ chd ++ che ++ chf ++ chg ++
                     chh ++ chi ++ chj ++ chk ++ chl ++ chm ++ chn ++
                     cho ++ chp ++ chq ++ chr ++ chs ++ cht ++ chu ++
                     chv ++ chw ++ chx ++ chy ++ chz ++ ch0 ++ ch1 ++
                     ch2 ++ ch3 ++ ch4 ++ ch5 ++ ch6 ++ ch7 ++ ch8 ++
                     ch9
DATA char <= ap(alphanumeric) ++ sp
DATA instream <= in(list list char)
DATA word <= wo(list alphanumeric)
DATA telegram <= te(list word)
DATA statistics <= st(num,truval)
DATA message <= me(telegram,statistics)

+++ getmessages(instream) <= list message
+++ gettels(instream) <= list telegram
+++ messagesof(list telegram) <= list message
+++ messof(telegram) <= message
+++ flatten(instream) <= list char
+++ wordsof(list char) <= list word
+++ telsof(list word) <= list telegram
+++ firstword(list char) <= word
+++ wtocl(word) <= list char
+++ wtoal(word) <= list alphanumeric
+++ firsttel(list word) <= telegram
+++ ttowl(telegram) <= list word
+++ statsof(telegram) <= statistics
+++ charge(list word) <= num
+++ okwl(list word) <= truval
+++ okw(word) <= truval
+++ length(list alfa) <= num
inf 4 =<     +++ num =< num <= truval
inf 4 -     +++ list alfa - list alfa <= list alfa
+++ zzzz <= word
+++ wstop <= word
+++ maxlen <= num

VAR INS : instream
VAR T : telegram     VAR TL : list telegram
VAR C : char     VAR CL : list char     VAR CLL : list list char
VAR W : word     VAR WL : list word
VAR A : alphanumeric     VAR AL : list alphanumeric
VAR N,N1 : num
VAR ALF : alfa     VAR ALFL,ALFL1 : list alfa

--- getmessages(INS) <= messagesof(gettels(INS))

--- messagesof(nil)   <= nil
--- messagesof(T::TL) <= messof(T)::messagesof(TL)

--- messof(T) <= me(T,statsof(T))
```

```
--- gettels(INS) <= telsof(wordsof(flatten(INS)))

--- flatten(in(nil::CLL))     <= flatten(in(CLL))
--- flatten(in((C::CL)::CLL)) <= C::flatten(in(CL::CLL))

--- wordsof(sp::CL)    <= wordsof(CL)
--- wordsof(ap(A)::CL) <= firstword(ap(A)::CL)
                :: wordsof(ap(A)::CL - wtocl(firstword(ap(A)::CL)))

--- firstword(sp::CL)    <= wo(nil)
--- firstword(ap(A)::CL) <= wo(A::wtoal(firstword(CL)))

--- wtocl(wo(nil))   <= nil
--- wtocl(wo(A::AL)) <= ap(A)::wtocl(wo(AL))

--- wtoal(wo(AL)) <= AL

--- telsof(W::WL) <= nil if firsttel(W::WL) = te(nil)
                  <= firsttel(W::WL) ::
                     telsof((W::WL - ttowl(firsttel(W::WL)))
                            - [zzzz]) ifnot

--- firsttel(W::WL) <= te(nil) if W = zzzz
                    <= te(W::ttowl(firsttel(WL))) ifnot

--- ttowl(te(WL)) <= WL

--- statsof(T) <= st(charge(ttowl(T)),okwl(ttowl(T)))

--- charge(nil)   <= 0
--- charge(W::WL) <= charge(WL) if W = wstop
                  <= succ charge(WL) ifnot

--- okwl(nil)   <= true
--- okwl(W::WL) <= okw(W) and okwl(WL)

--- okw(W) <= length(wtoal(W)) =< maxlen

--- length(nil) <= 0
--- length(ALF::ALFL) <= succ length(ALFL)

--- 0 =< N1 <= true
--- succ N =< 0 <= false
--- succ N =< succ N1 <= N =< N1

--- nil - ALFL1 <= nil
--- ALFL - nil <= ALFL
--- ALF::ALFL - ALF::ALFL1 <= ALFL - ALFL1

--- zzzz <= wo([chz,chz,chz,chz])

--- wstop <= wo([chs,cht,cho,chp])

--- maxlen <= 12
```

END

This protoprogram is straightforward but not at all efficient . The input goes through four distinct passes (instream -> list char -> list word -> list telegram -> list message) and the passes are themselves very inefficient. We will aim to transform this into an efficient single-pass solution.

5.3.4 Transforming To Efficient Version

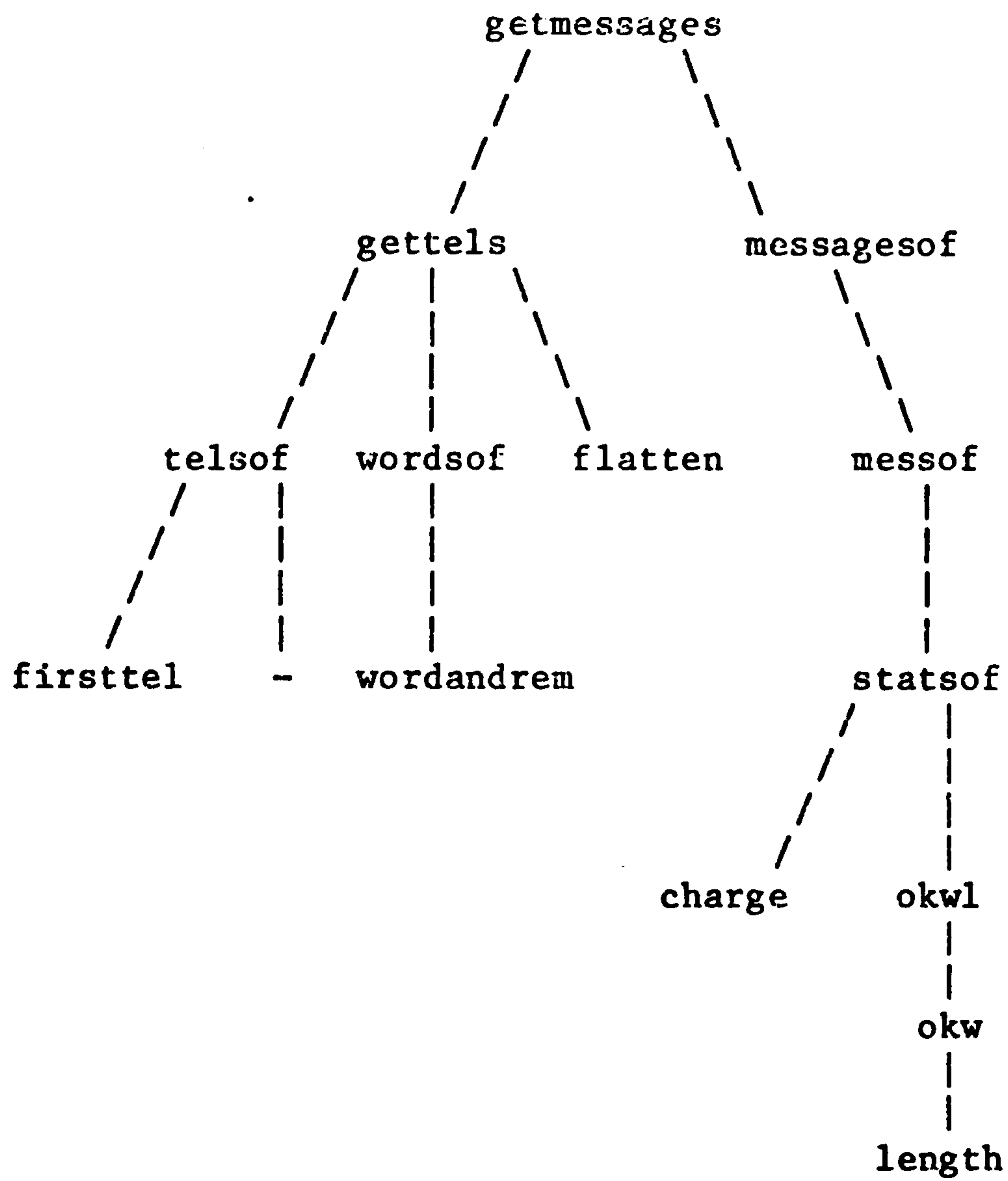
Adopting the simple strategy outlined in section 5.2 suggests we tackle the transformation of the above protoprogram in the following stages:

(i) Improve wordsof(ap(A)::CL) by

(a) Combining clremaining(CL) <= CL -- wtocl(firstword(CL))

(b) Tupling wordandrem(CL) <= <firstword(CL),clremaining(CL)>

This changes the program structure to the following:

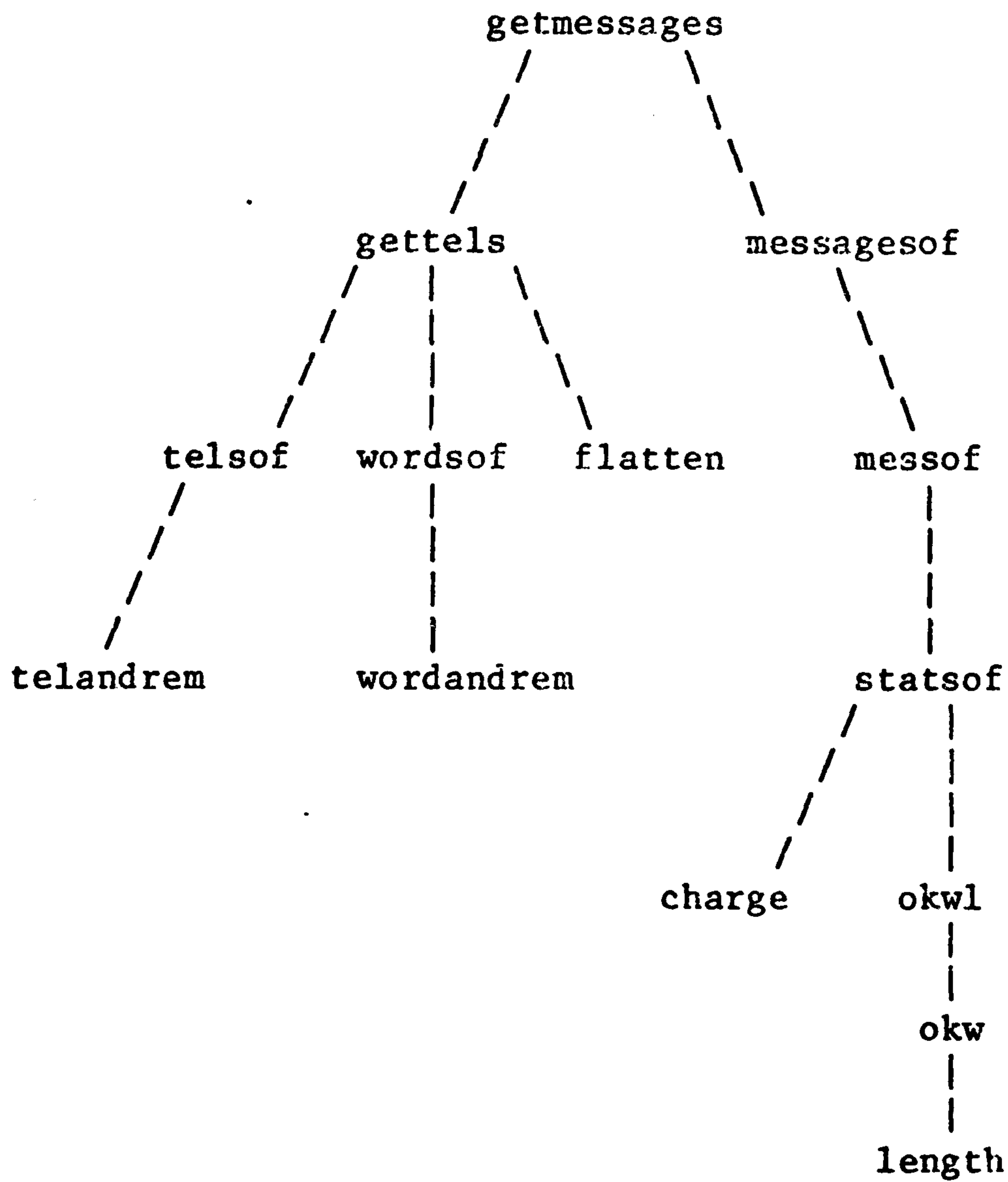


(ii) Improve telsof(W::WL) by

(a) Combining wlremaining(WL) <= (WL - ttowl(firsttel(WL)))

- zzzz

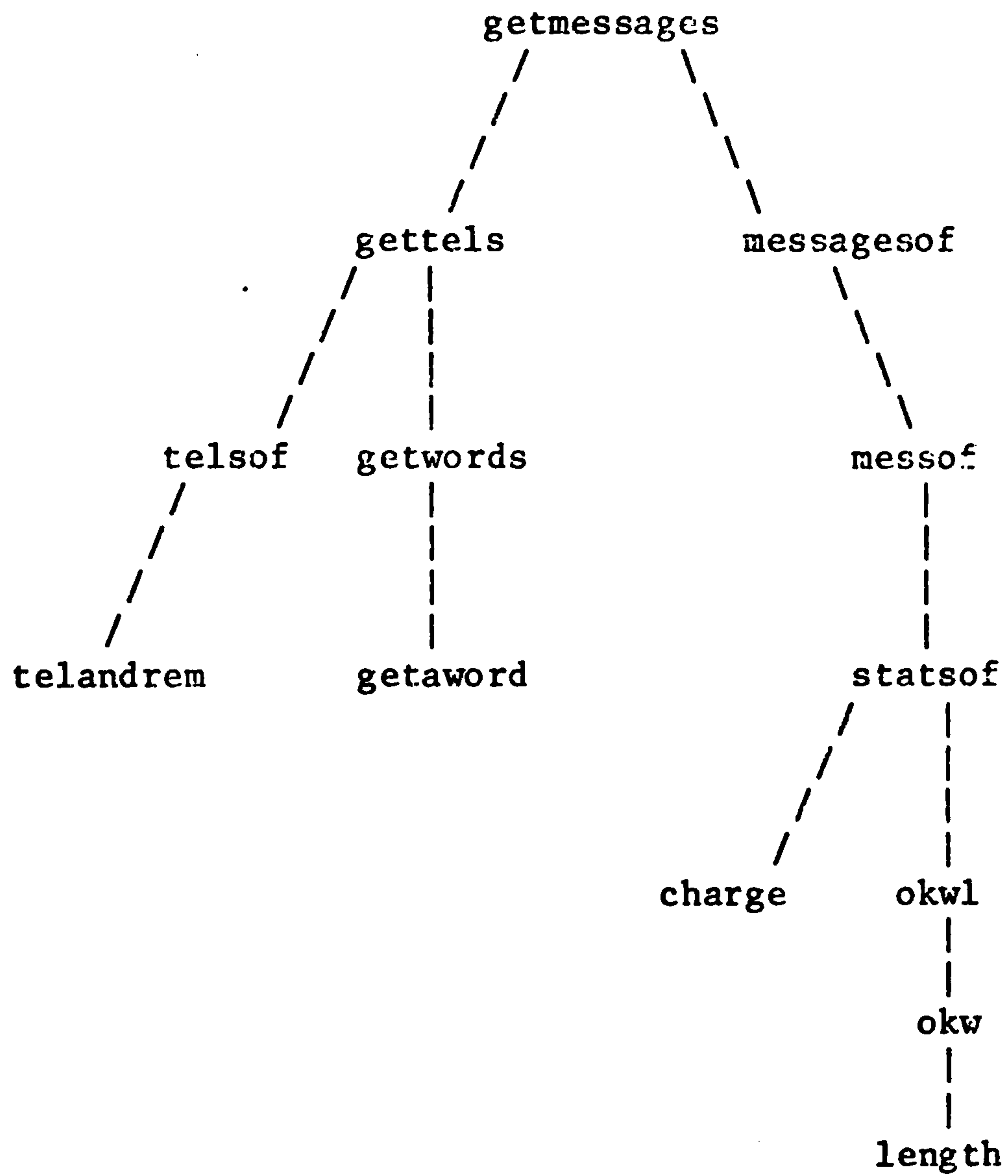
(b) Tupling telandrem(WL) <= <firsttel(WL),wlremaining(WL)>

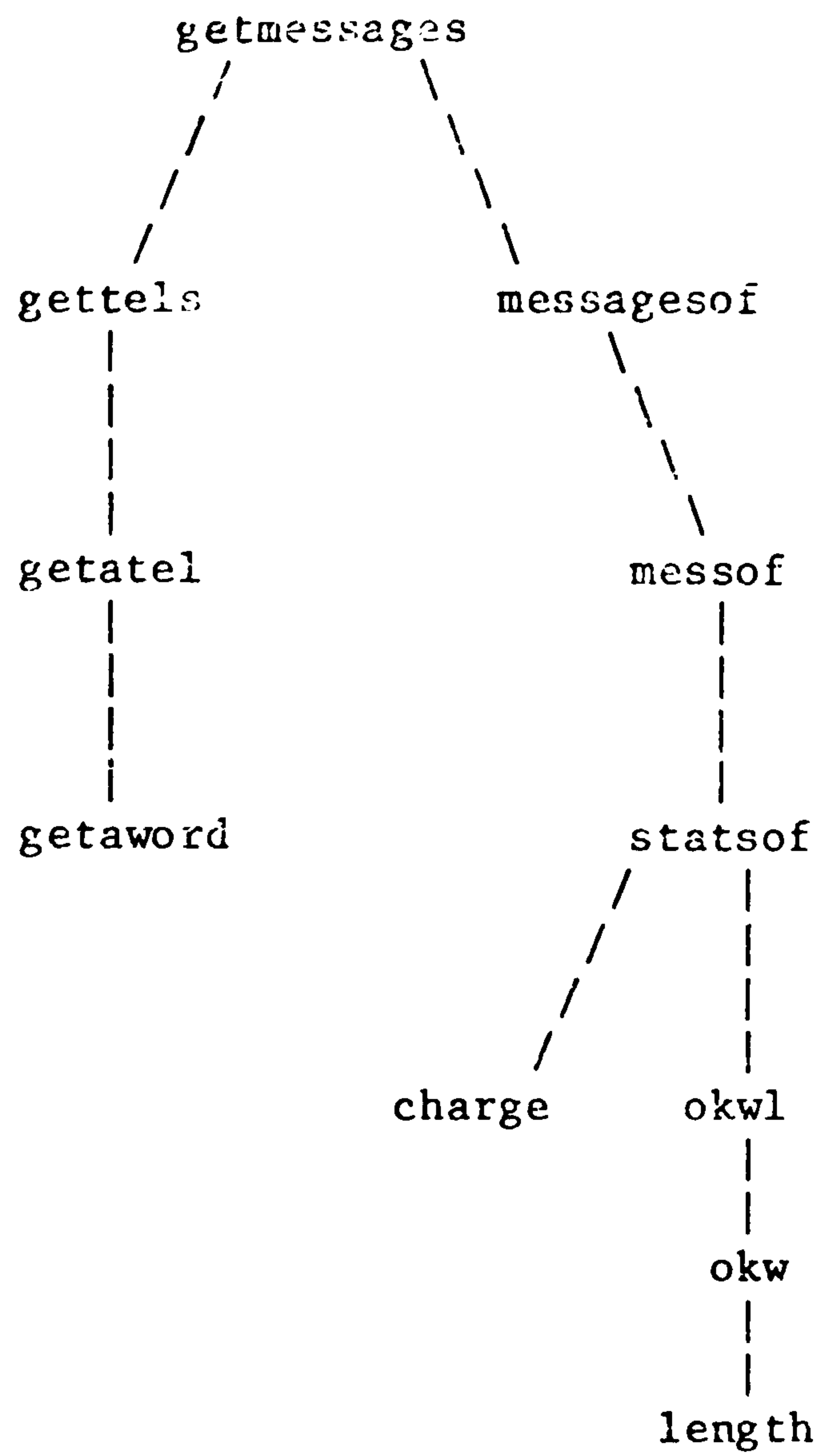


(iii) Improve gettels by

(a) Combining getwords(INS) <= wordsof(flatten(INS))

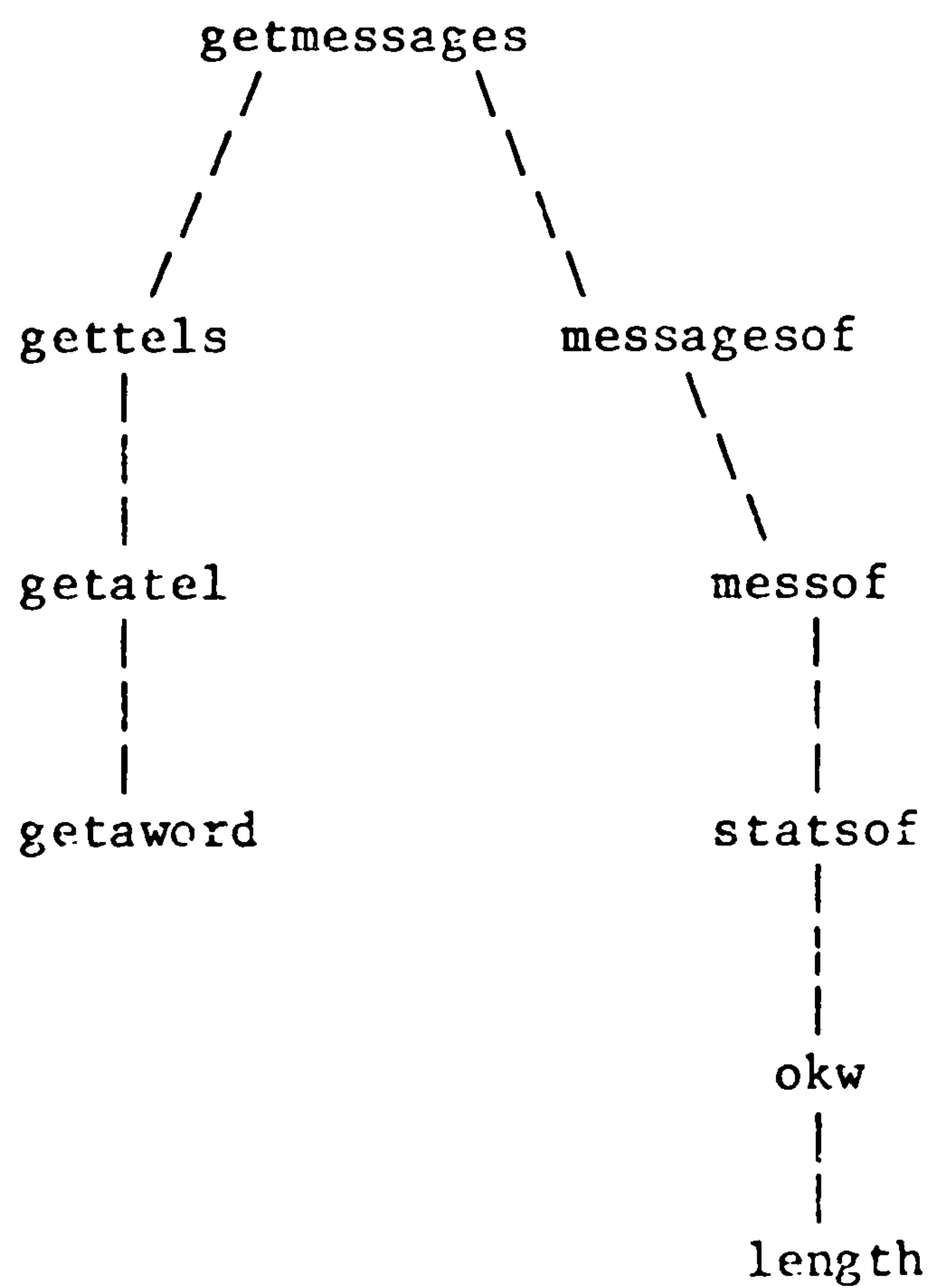
(b) Combining gettels(INS) <= telsof(getwords(INS))





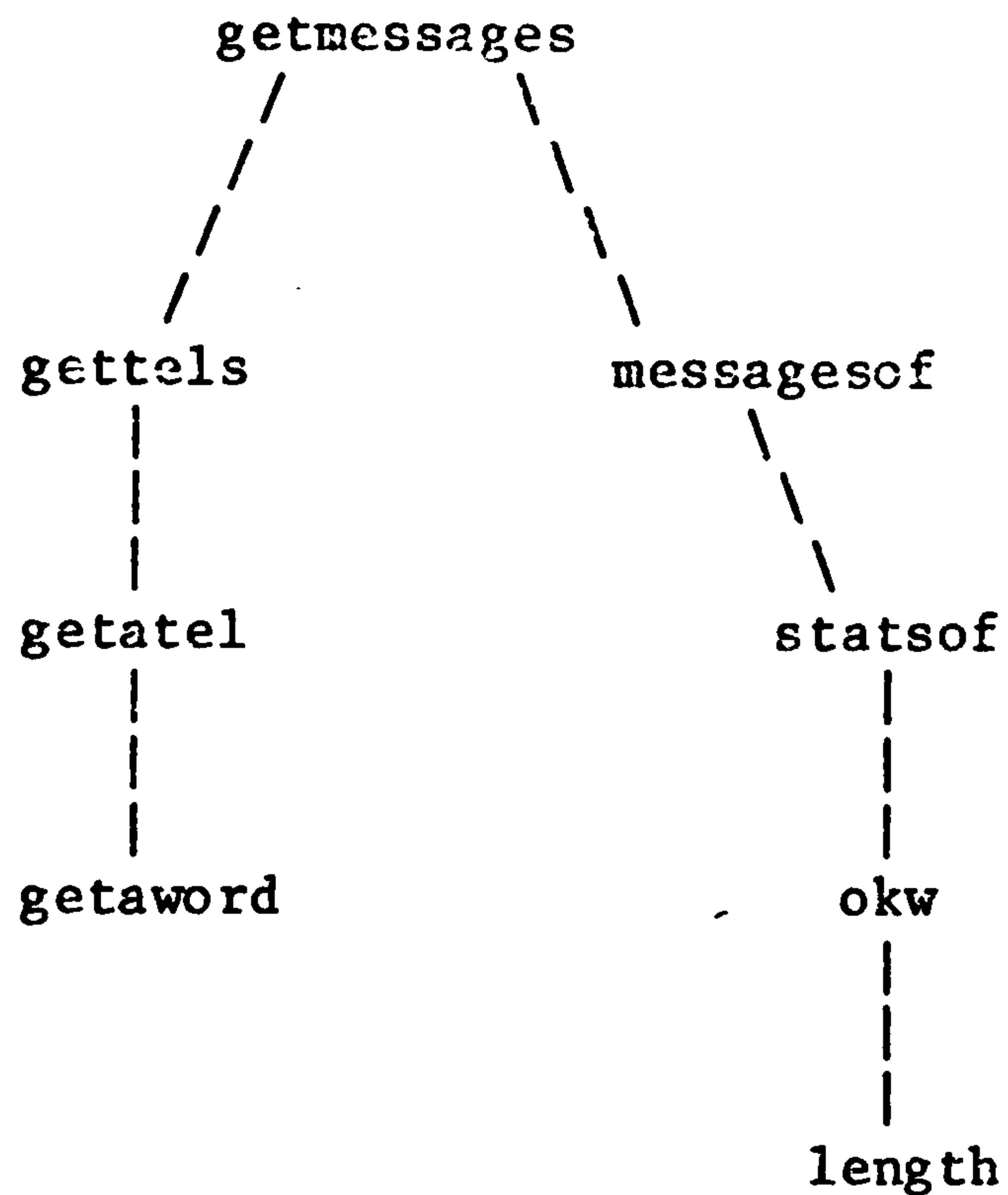
(iv) Improve statsof by

Combining statsof(T) <= st(charge(ttowl(T)),okwl(ttowl(T)))



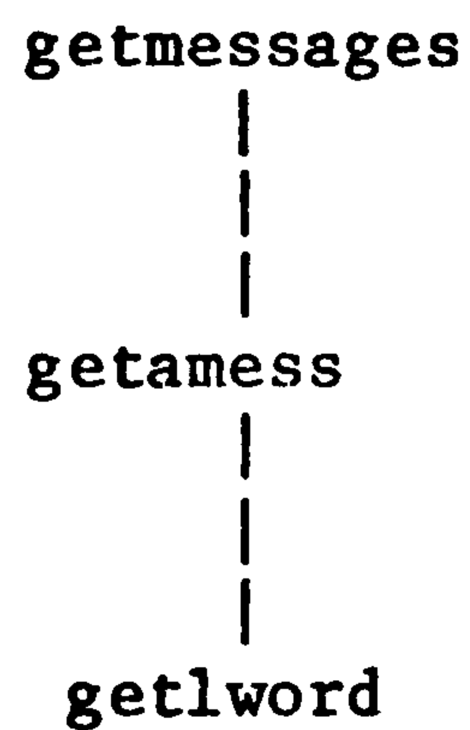
(v) Simplify messagesof(T::TL) by expanding out definition of messof(T)

(note that this is neither a combination nor a tuple; the effect of this step is merely to cause a small simplification in the program structure.)



(vi) Improve getmessages by

Combining getmessages(INS) <= messagesof(gettels(INS))



(vii) Convert to an iterative form suitable for conversion into an imperative language.

Now I present the commands for each stage as given to the ZAP system to carry out the transformations.

(i) Improve wordsof

Since steps (a) and (b) are both small, we do both at once;

```
INTRODUCE VAR CLREM : list char END
CONTEXT /// 'improve wordsof @
   UNFOLD wordsof
   USING wtocl wordsof
         RESTRICTED firstword -
   TRANSFORM
      GOAL wordsof(ap(A)::CL) <= W::wordsof(CLREM)
                where <W,CLREM> == &&wordandrem(ap(A)::CL)
   END
END

CONTEXT /// 'redefine wordandrem recursively @
   UNFOLDALL wordandrem
   USING wtocl wtoal wordandrem
   LEMMAS --- wo(wtoal(W)) <= W
   TRANSFORM
      GOAL wordandrem(sp::CL)
      GOAL wordandrem(ap(A)::CL) <= $$(A,wordandrem(CL))
   END
   DELETE wordandrem(CL)
END
```

(ii) Improve TELSOF

Since steps (a) and (b) are both small, again we do both at once;

```
INTRODUCE VAR WLREM : list word END
CONTEXT /// 'improve telsof @
   UNFOLD telsof
   USING ttowl telsof
         RESTRICTED firsttel -
   TRANSFORM
      GOAL telsof(W::WL) <= $$(T,telsof(WLREM))
            where <T,WLREM> == &&telandrem(W::WL)
   END
END

CONTEXT /// 'redefine telandrem recursively @
   UNFOLDALL telandrem
   USING ttowl telandrem
   TRANSFORM
      GOAL telandrem(W::WL) <= $$(W,WL,telandrem(WL))
   END
   DELETE telandrem(WL)
```

```
END
```

(iii) Improve gettels

This is the first transformation to involve the data type instream, which represents a buffered stream of input. For this data type we anticipate transformations involving cases

```
in(nil::CLL)
in((sp::CL)::CLL)
in((ap(A)::CL)::CLL)
```

The first of these represents reaching the end of the current input buffer, so we would expect to continue processing the remainder of the stream, i.e. in(CLL).

The last two cases represent encountering a space or alphanumeric within the current input buffer. Since we expect our transformed algorithm to work character by character, we anticipate processing would continue with the remainder of the current buffer and stream, i.e. in(CL::CLL)

The TYPEINFO command allows us to present such intuition as information about the data type instream:

```
TYPEINFO
   INS <= in(nil::CLL), in(CLL)
       <= in((sp::CL)::CLL), in(CL::CLL)
       <= in((ap(A)::CL)::CLL), in(CL::CLL)
```

(a) Combining getwords(INS) <= wordsof(flatten(INS))

```
CONTEXT /// 'improve gettels by combining wordsof & flatten @
   UNFOLD gettels
   USING telsof RESTRICTED gettels wordsof flatten
   TRANSFORM
      GOAL gettels(INS) <= telsof(&&getwords(INS))
   END
END

INTRODUCE VAR REMINS : instream END
CONTEXT /// 'redefine getwords recursively @
```

```
   UNFOLDALL getwords
   USING wtoal RESTRICTED getwords wordsof flatten wordandrem
   TRANSFORM
      GOAL getwords(in(nil::CLL)) <= auto
      GOAL getwords(in((sp::CL)::CLL)) <= auto
      GOAL getwords(in((ap(A)::CL)::CLL)) <=
                              $$(A,W) :: getwords(REMINS)
               where <W,REMINS> == &&getaword(in(CL::CLL))
   END
   DELETE getwords(INS)
END

CONTEXT /// 'simplify by folding up @
   USING wtoal RESTRICTED getwords getaword
   UNFOLDALL getwords
   TRANSFORM
      GOAL getwords(in((ap(A)::CL)::CLL)) <= $$(W,getwords(REMINS))
            where <W,REMINS> == getaword(in((ap(A)::CL)::CLL))
   END
END


CONTEXT   /// 'redefine getaword recursively @
   UNFOLDALL getaword
   USING wtoal getaword
   LEMMAS --- iflatten(flatten(INS)) <= INS
          --- iflatten(sp::flatten(in(CL::CLL))) <= in((sp::CL)::CLL)
   TRANSFORM
      GOAL getaword(CASESOF INS) <= auto
   END
   DELETE getaword(INS)
END
```

Notice that we have used the lemmas

```
iflatten(flatten(INS)) <= INS

iflatten(sp::flatten(in(CL::CLL))) <= in((sp::CL)::CLL)
```

iflatten is the inverse of flatten, introduced by the system during the definition of getaword. The first lemma simply states the inverse relationship. The second is required because the unfolding mechanism would otherwise reduce

```
iflatten(flatten(in((sp::CL)::CLL)) to

iflatten(sp::flatten(in(CL::CLL)))
```

at which point it would be stuck (instead of attaining in((sp::CL)::CLL)).

(b) Combining gettels(INS) <= telsof(getwords(INS))

```
INTRODUCE VAR REMINS1 : instream
          VAR T,T1 : telegram
END
CONTEXT /// 'redefine gettels recursively @
   UNFOLDALL gettels
   USING RESTRICTED gettels
   TRANSFORM
      GOAL gettels(in(nil::CLL)) <= auto
      GOAL gettels(in((sp::CL)::CLL)) <= auto
   END
END

CONTEXT /// 'now for the in((ap(A)::CL)::CLL) case @
   UNFOLD gettels getwords telsof telandrem
   USING ttowl wtoal RESTRICTED gettels getwords telandrem getaword
 telsof
   LEMMAS --- te(W::WL) = te(NIL) <= false
   TRANSFORM
      GOAL gettels(in((ap(A)::CL)::CLL)) <=
               cond($$(W),NIL,T::gettels(REMINS))
                where <T> == <te(W::ttowl(T1))>
                 where <T1,REMINS> == &&getatel(REMINS1)
                  where <W,REMINS1> == getaword(in((ap(A)::CL)::CLL))
   END
   DELETE gettels(INS)
END

CONTEXT /// 'simplify by folding up @
   USING ttowl RESTRICTED gettels getatel
   UNFOLDALL gettels
   LEMMAS --- te(W::WL) = te(NIL) <= false
   TRANSFORM
      GOAL gettels(in((ap(A)::CL)::CLL)) <=
             cond(T=te(NIL),NIL,T::gettels(REMINS))
               where <T,REMINS> == getatel(in((ap(A)::CL)::CLL))
   END
END


INTRODUCE VAR REMINS1,REMINS2 : instream END
CONTEXT /// 'redefine getatel recursively@
   UNFOLDALL getatel ZZZZ
   USING ttowl getatel RESTRICTED getaword
   LEMMAS --- igetwords(getwords(INS)) <= INS
   TRANSFORM
      GOAL getatel(in(NIL::CLL)) <= auto
      GOAL getatel(in((sp::CL)::CLL)) <= auto
      GOAL getatel(in((ap(A)::CL)::CLL)) <=
                               $$(ZZZZ,W,REMINS1,REMINS2,T)
           where <T,REMINS2> == getatel(REMINS1)
            where <W,REMINS1> == getaword(in((ap(A)::CL)::CLL))
   END
END
```

Having completed this stage, the overall structure of our efficient solution has been formed. In this structure the outermost function builds up the entire result using another function to build single telegrams at once, and this in turn uses a smaller function to build up individual words. Combining the statistics production with this will merely cause the functions to return additional arguments without altering this structure.

(iv) Combining statsof(T) <= st(charge(ttowl(T)),okwl(ttowl(T)))

```
INTRODUCE VAR OK : truval
   +++ ston(statistics) <= num      --- ston(st(N,OK)) <= N
   +++ stook(statistics) <= truval --- stook(st(N,OK)) <= OK
END

CONTEXT
   USING statsof okw and ston stook
   UNFOLD statsof charge okwl
   TRANSFORM
      GOAL statsof(te(nil))
      GOAL statsof(te(W::WL)) <= st($$(W,CH),$$(W,OK))
                where <N,OK> == <ston(S),stook(S)>
                where <S> == <statsof(te(WL))>
   END
   DELETE statsof(T)
END
```

(v) Simplifying messagesof(T::TL)

```
CONTEXT
   USING messagesof statsof
   UNFOLD messagesof messof
   TRANSFORM
      GOAL messagesof(T::TL) <= $$(T,statsof(T),messagesof(TL))
   END
END
```

(vi) Combining getmessages(INS) <= messagesof(gettels(INS))

```
INTRODUCE  VAR S : statistics
END

CONTEXT /// 'redefine getmessages recursively @
   UNFOLDALL getmessages
   USING RESTRICTED getmessages
   TRANSFORM
```

```
      GOAL getmessages(in(nil::CLL)) <= auto
      GOAL getmessages(in((sp::CL)::CLL)) <= auto
   END
END

INTRODUCE VAR WLEN : num
CONTEXT /// 'now for the ap(A) case @
   UNFOLD getmessages messagesof gettels
   USING RESTRICTED getmessages getatel statsof
   TRANSFORM
      GOAL getmessages(in((ap(A)::CL)::CLL)) <=
           cond($$(T),NIL,me(T,S)::getmessages(REMINS))
            where <T,S,REMINS> == &&getamess(in((ap(A)::CL)::CLL))
   END
   DELETE getmessages(INS)
END


CONTEXT /// 'redefine getamess recursively@
   UNFOLDALL getamess
   USING RESTRICTED getamess
   TRANSFORM
      GOAL getamess(in(NIL::CLL)) <= auto
      GOAL getamess(in((sp::CL)::CLL)) <= auto
   END
END

CONTEXT /// 'now for the ap(A) case @
   UNFOLD getamess ston stook getatel statsof ttowl charge okwl okw
   USING ston stook =< ttowl length wtoal and
         RESTRICTED getamess getaword
   TRANSFORM
      GOAL getamess(in((ap(A)::CL)::CLL)) <=
             $$(WLEN,W,REMINS1,REMINS,ston(S),stook(S),T)
              where <T,S,REMINS> == getamess(REMINS1)
               where <W,REMINS1,WLEN> ==
                                    &&getlword(in((ap(A)::CL)::CLL))
   END
   DELETE getamess(INS)
END

CONTEXT /// 'now transform getlword @
   UNFOLDALL getlword
   USING getlword
   TRANSFORM
      GOAL getlword(CASESOF INS) <= AUTO
   END
   DELETE getlword(INS)
END
```

(vii) Convert to iterative form

```
INTRODUCE
   +++ agetmessages(instream,list message) <= list message
   +++ agetamess(instream,list word,num,truval) <=
                              tuple2(message,instream)
   +++ agetamess(instream,list char,num) <=
                              tuple3(word,instream,num)

   --- agetmessages(INS,ML) <= ML <> getmessages(INS)

   --- agetamess(INS,WL,N,OK) <=
            <me(te(WL<>ttowl(mtot(M))),
                st(N+ston(mtos(M)),OK and stook(mtos(M)))),
             REMINS>
          where <M,REMINS> == getamess(INS)

   --- agetlword(INS,CL,N) <=
             <wo(CL<>wtocl(W)),REMINS,N+N1>
          where <W,REMINS,N1> == getaword(INS)
END


CONTEXT
   UNFOLD getmessages agetmessages <>
   USING RESTRICTED agetmessages
   TRANSFORM
      GOAL getmessages(INS) <= agetmessages(INS,nil)
   END
END

CONTEXT /// 'redefine agetmessages in terms of
             itself and agetamess @
   UNFOLD agetmessages getmessages <> + and agetamess
   USING mtot <>
         RESTRICTED agetmessages agetamess
   TRANSFORM
      GOAL agetmessages(in(nil::CLL)) <= auto
      GOAL agetmessages(in((sp::CL)::CLL)) <= auto
      GOAL agetmessages(in((ap(A)::CL)::CLL) <=
              $$(M,ML,agetmessages(REMINS,ML<>[M]))
        where <M,REMINS> ==
                 agetamess(in((ap(A)::CL)::CLL,nil,0,true)
   END
END

CONTEXT /// 'redefine agetamess in terms of
             itself and agetlword @
   UNFOLD agetamess getamess agetlword <> + and =<
   USING + and <> =< ttowl ston wtoal stook
        RESTRICTED agetamess agetlword
   TRANSFORM
      GOAL agetamess(in(nil::CLL),WL,N,OK) <= auto
      GOAL agetamess((sp::CL)::CLL),WL,N,OK) <= auto
      GOAL agetamess(in((ap(A)::CL)::CLL),WL,N,OK) <=
           $$(W,REMINS,WLEN,WL,N,OK,
              agetamess(REMINS,WL<>[W],$$(W,N),$$(WLEN,MAXLEN,OK)))
              where <W,REMINS,WLEN> ==
```

```
                          agetlword(in((ap(A)::CL)::CLL),nil,0)
   END
END

CONTEXT /// 'redefine agetlword recursively @
   UNFOLDALL agetlword
   USING <> and + =<
        RESTRICTED agetlword
   TRANSFORM
      GOAL agetlword(in(nil::CLL),CL1,N) <= auto
      GOAL agetlword(in((sp::CL)::CLL),CL1,N) <= auto
      GOAL agetlword(in((ap(A)::CL)::CLL),CL1,N) <=
               agetlword(in(CL::CLL),CL1<>[ap(A)],succ N)
   END
END

DELETE agetmessages(INS,ML),
       agetamess(INS,WL,N,OK),
       agetlword(INS,CL,N)
```

5.3.5 Final Program

The final program works by building up individual words (using agetlword) and individual messages (using agetamess) in a <u>single</u> pass, as it works its way through the input. The length of a word is counted as the word is extracted, and the statistics for each telegram are amassed as each telegram is extracted.

This is far more efficient than the original program, which consisted of four major passes through the input, going from instream to list char to list word to list telegram to list message. Some of these passes were themselves very inefficient.

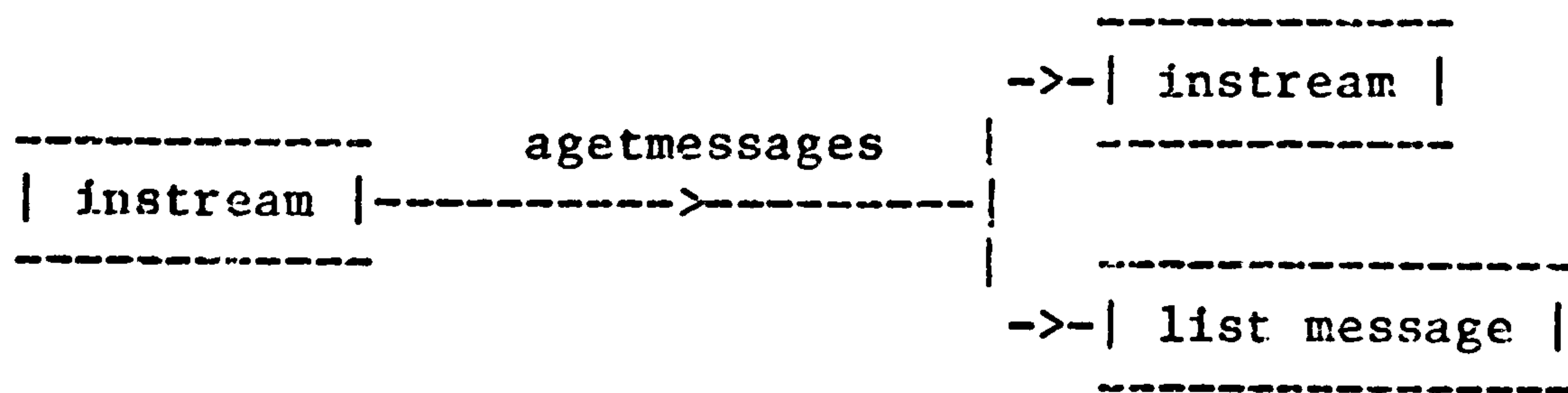
```
--- getmessages(INS) <= agetmessages(INS,nil)

--- agetmessages(in(nil::CLL),ML) <=
                    agetmessages(in(CLL),ML)

--- agetmessages(in((sp::CL)::CLL),ML) <=
                    agetmessages(in(CL::CLL),ML)

--- agetmessages(in((ap(A)::CL)::CLL,ML) <=
           cond(mtot(M) = te(nil),
                ML,
```

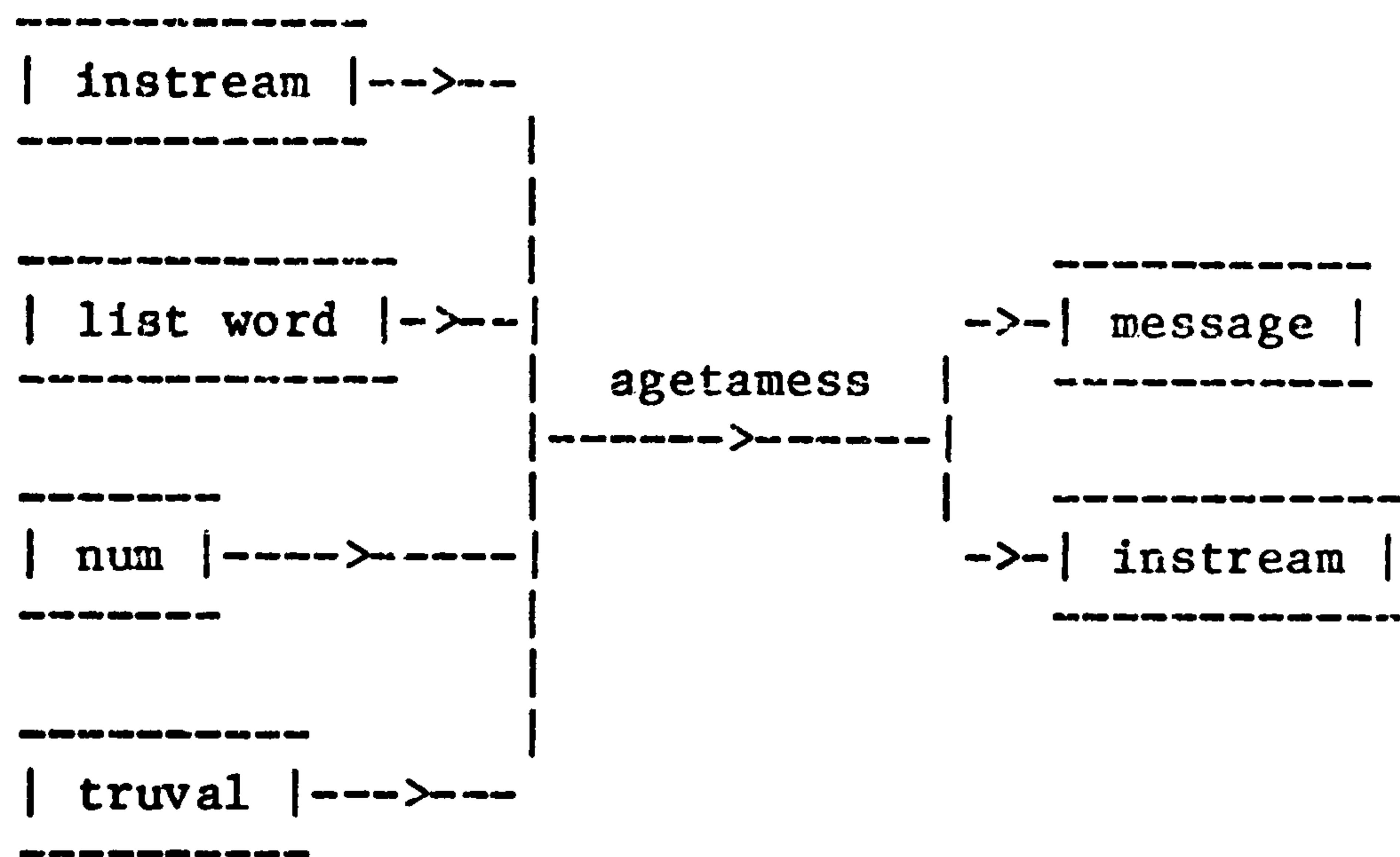
```
                agetmessages(REMINS,ML<>[M]))
        where <M,REMINS> ==
                agetamess(in((ap(A)::CL)::CLL),nil,0,true)

--- agetamess(in(nil::CLL),WL,N,OK) <= agetamess(in(CLL),WL,N,OK)

--- agetamess(in((sp::CL)::CLL),WL,N,OK) <=
                       agetamess(in(CL::CLL),WL,N,OK)

--- agetamess(in((ap(A)::CL)::CLL),WL,N,OK) <=
            cond(W=ZZZZ,
                <me(te(WL),st(N,OK)),REMINS>,
                 agetamess(REMINS,WL<>[W],
                           cond(W=WSTOP,N,succ N)
                           (WLEN =< MAXLEN) and OK))
        where <W,REMINS,WLEN> ==
                 agetlword(in((ap(A)::CL)::CLL),nil,0)

--- agetlword(in(nil::CLL),CL1,N) <= agetlword(in(CLL),CL1,N)

--- agetlword(in((sp::CL)::CLL),CL1,N) <=
                 <wo(CL1),in((sp::CL)::CLL),N>

--- agetlword(in((ap(A)::CL)::CLL),CL1,N) <=
                 agetlword(in(CL::CLL),CL1<>[ap(A)],succ N)
```

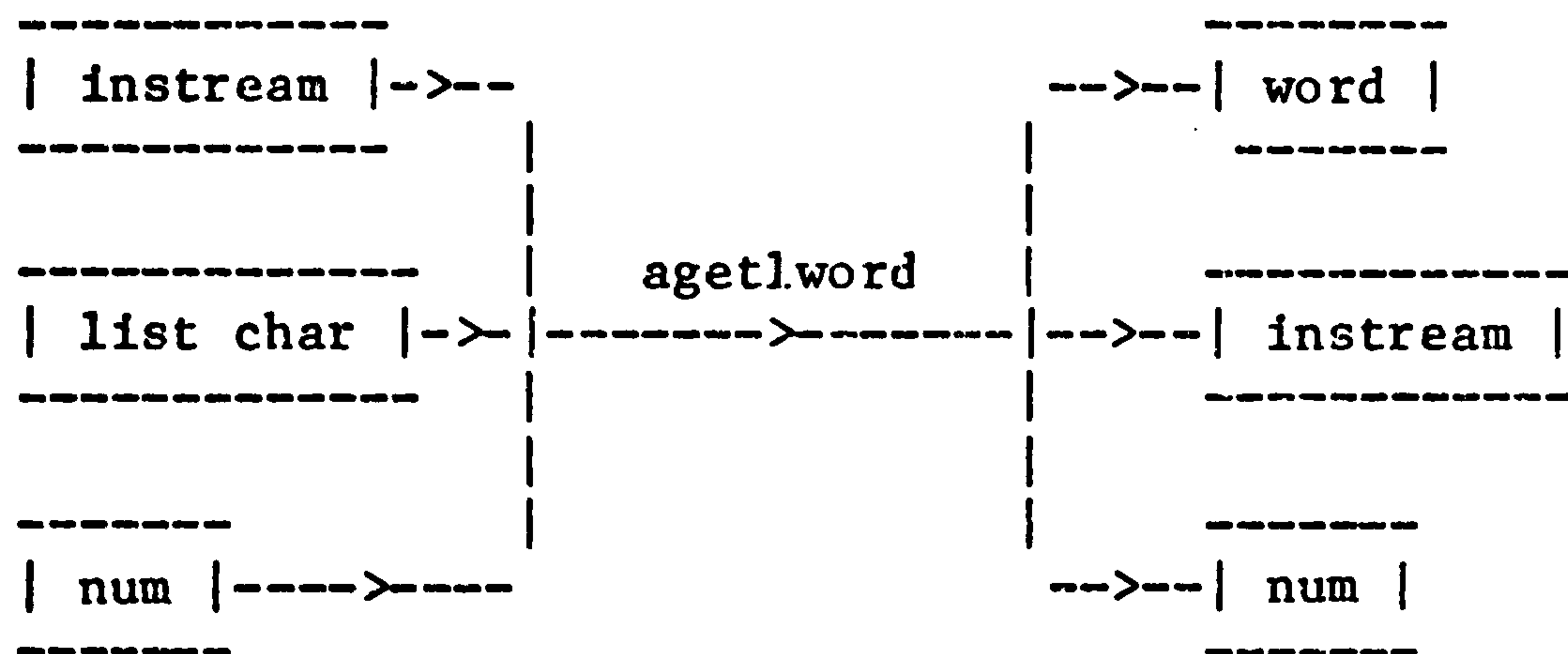
The structure of this final program is simple:

```
agetmessages
     |
     |
 agetamess
     |
     |
 agetlword
```

However, these functions are performing several operations at once in order that the final program be a one-pass solution. The diagrams of the input and output types for these functions illustrate the complexity:







The final NPL program requires converting into an imperative language. As an illustration of this, the innermost function, agetlword, might convert to the following program in some ALGOL-like language:

```
PROCEDURE   agetlword( INSTREAM VALUE RESULT ins,
                       INTEGER VALUE RESULT wlen,
                       LIST(CHARACTER) VALUE acl,
                       WORD RESULT W);
BEGIN
   WHILE  ( WHILE hd(ins) = nil
            DO ins := tl(ins) OD;
            hd(hd(ins)) /= space )
   DO acl := acl <> [ hd(hd(ins)) ];
      wlen := 1 + wlen;
      ins := tl(hd(ins)) :: tl(ins)
   OD;
   w := wo(acl)
END;
```

Some improvements at this level are still possible. For example, acl is passed to the procedure agetlword, and within a while loop a character is appended to its end. This could be more efficiently done by maintaining a pointer to the end of acl, and destructively appending to the end of the list.

Such improvements do not alter the structure of the solution, and I have not investigated them. My main concern has been with the change from the naive solution to the very different structure of an efficient solution.

5.3.6 Modification Of Telegram Problem

One of the hoped for virtues of the transformational approach to programming is that program modification may be carried out easily and reliably. To investigate this, I make a small change in the original specification of the telegram problem, and then see how the transformation process must be adjusted to accomodate this change.

The alteration I make is to charge double for overlength words. Modification of the initial program is easy; function charge is designed to charge one unit for each word other than "STOP".

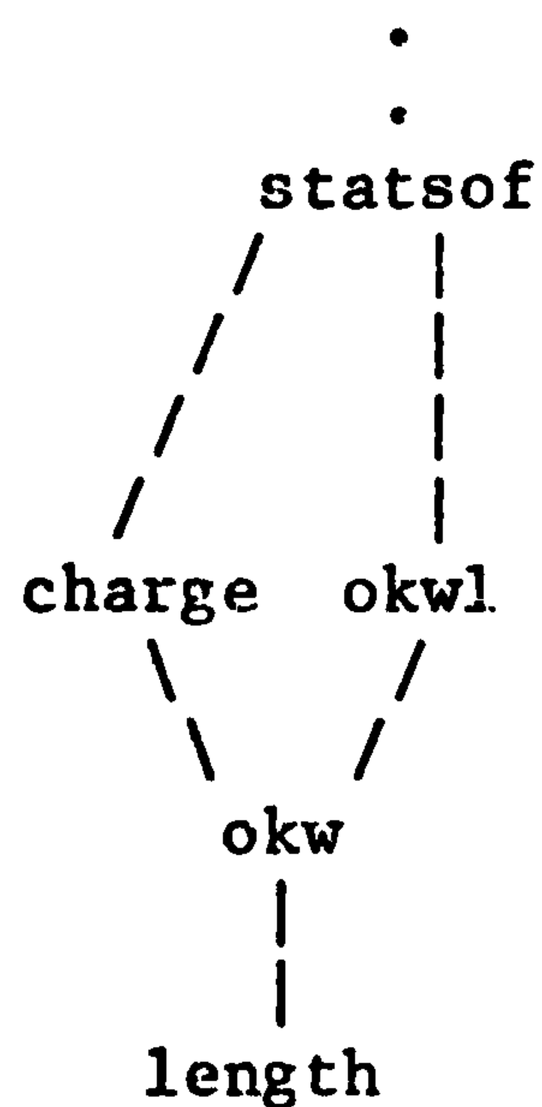
```
--- charge(W::WL) <= charge(WL) if W=WSTOP
                  <= succ charge(WL) ifnot
```

This must be modified to

```
--- charge(W::WL) <= charge(WL) if W=WSTOP
                  <= succ charge(WL) if okw(W)
                  <= succ succ charge(WL) ifnot
```

making use of function okw to check words for admissible length.

Thus the change in program structure is to cause charge to now make use of okw:



Now to consider how the transformation process is effected:

Stages (i), (ii) and (iii) remain unchanged, since they do not concern the altered portion of program structure.

Stage (iv), combining statsof(T) <=

```
st(charge(ttowl(T)),okwl(ttowl(T)))
```

is the first to concern the altered portion. The transformation commands suffice as they are, however the recursive definition of statsof they give rise to is slightly different.

Stage (v) is unaffected since statsof is not unfolded during it.

Stage (vi) is the next to make use of the definition of statsof - in redefining the ap(A) case of getamess. Once again, the existing transformation commands suffice unchanged, the modified statsof inducing a modification to getamess.

Stage (vii) is the conversion to iterative form. Here the redefinition of agetamess, because it is defined in terms of the now modified getamess, potentially needs altering. In fact a small change in the transformation commands _is_ required here:

We have

```
   GOAL agetamess(in((ap(A)::CL)::CLL),WL,N,OK) <=
          $$(W,REMINS,WLEN,WL,N,OK,
             agetamess(REMINS,WL<>[W],$$(W,N),$$(WLEN,MAXLEN,OK)))
               where <W,REMINS,WLEN> ==
                 agetlword(in((ap(A)::CL)::CLL),nil,0)
```

The third argument of the call to agetamess, i.e. $$(W,N), is the accumulating charge for the current telegram. Our modified charging algorithm now takes account of the length of words, so this argument must be expanded to $$(W,N,WLEN). WLEN is the length of the current word, returned from a call to agetlword.

The effect on the final NPL program is to change one of the equations of agetamess to:

```
--- agetamess(in((ap(A)::CL)::CLL),WL,N,OK) <=
       cond(W=ZZZZ,
            <me(te(WL),st(N,OK)),REMINS>,
            agetamess(REMINS,WL<>[W],
                      cond(W=WSTOP,N,
**change-->             cond(WLEN =< MAXLEN,succ N, succ succ N)),
                      (WLEN =< MAXLEN) and OK))
           where <W,REMINS,WLEN> ==
                 agetlword(in((ap(A)::CL)::CLL),nil,0)
```

Thus for this particular modification the change in the final, efficient program is relatively small, and a competent programmer could no doubt have made the appropriate change in the efficient program directly. To do so would require pinpointing the location within the efficient program where the change must be made, hence the programmer would have to understand how the efficient code functioned, a requirement we would like to avoid. It is encouraging to see that the propogation of the modification through the transformation has been easy. The well-structured transformation pinpoints the areas potentially effected, and the transformation patterns are sufficiently powerful as to accomodate the modification without the need for adjustment until the very last stage.

5.4 SIMPLE COMPILER

This example consists of the second phase of a simple compiler taking abstract syntax trees to machine code. The source statements consist of assignments, while statements, if-then-elses and blocks. Expressions within the source statements consist of either a variable, or an operator applied to a list of expressions. The NPL representation of these is the following:

```
DATA variable <= varible

DATA opr <= ops

DATA sstatement <= asst(variable,expression)

                ++ whst(expression,sstatement)

                ++ ifthenelse(expression,sstatement,sstatement)

                ++ bl(list variable,sstatement)

                ++ sstatement $ sstatement;

     expression <= expr(variable) ++ apply(opr,list expression)
```

The target machine code assumes a stack machine with instructions to load a value from an address onto the stack, store the value on the top of the stack in some address, jump to an address, conditionally jump to an address dependent upon the value on the top of the stack, apply an operator (which is assumed to take off the appropriate number of values from the stack, returning the answer onto the top of the stack), and finally a nonop instruction (to do nothing). A machine code program is a list of these instructions. The NPL data definitions to represent these are

```
DATA minstruction <= mload(address) ++ mstr(address)

                  ++ mcondjump(address) ++ mjump(address)

                  ++ mapp(opp) ++ mnonop
```

```
DATA mcode <= mpr(list minstruction)
```

Addresses in the machine are represented by

```
DATA address <= ad(num)
```

with its own function (incadd) to increment an address to get the next in store.

e.g.

The program written conventionally as

```
BEGIN
   VARS KO K1 K2;
   K2 := OP4(K1);
   BEGIN
      VARS K3,KO;
      WHILE KO DO
         KO := OP2(KO,K2,K3)
      OD;
   END;
   IF OP3(K1)
   THEN K1 := KO
   ELSE KO := K1
   FI
END
```

would be represented by

```
bl([KO,K1,K2],
   asst(K2,apply(OP4,[expr(K1)]))
   $
   bl([K3,KO],
      whst(expr(KO),
           asst(KO,apply(OP2,
                  [expr(KO),expr(K2),expr(K3)]))))
   $
   ifthenelse(apply(OP3,[expr(K1)]),
              asst(K1,expr(KO)),
              asst(KO,expr(K1)))
  )
```

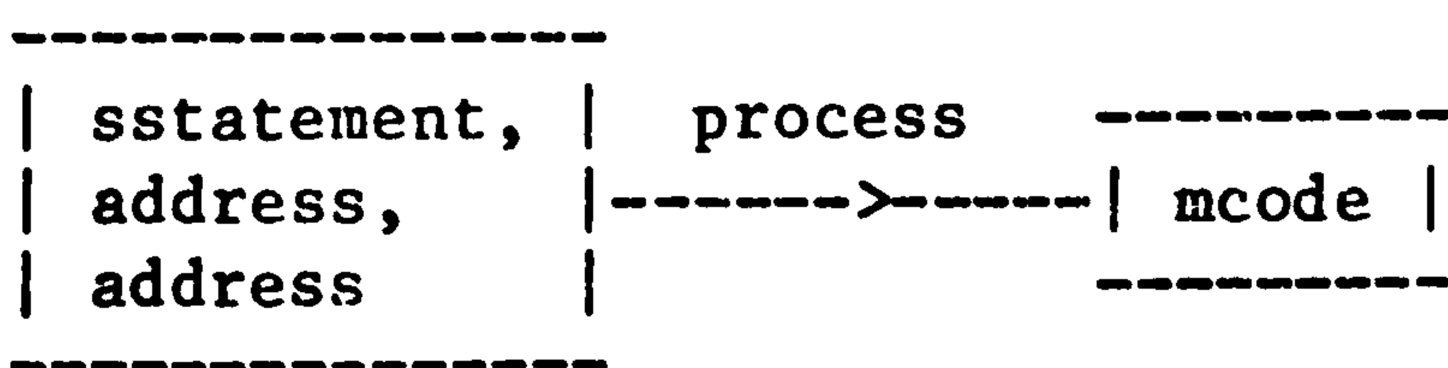
allocating space for code from address 0, and space for variables from address 25, compiling the above gives (I omit to put ad() around the number of each address below, for clarity):

address	instruction
0	mload 26
1	mapp OP4
2	mstr 27
3	mnonop
4	mload 29
5	mcondjump 12
6	mload 29
7	mload 27
8	mload 28
9	mapp OP2
10	mstr 29
11	mjump 3
12	mnonop
13	mload 26
14	mapp OP3
15	mcondjump 19
16	mload 26
17	mstr 25
18	mjump 22
19	mnonop
20	mload 25
21	mstr 26
22	mnonop

5.4.1 Design Of Protoprogram

The NPL protoprogram I produce serves as my exact problem specification.

The overall process is one of converting the source statement input into machine code, allocating space for the code from some given adddress, and allocating space for variables from another address.



This splits into two tasks - the first is to convert the source statements into an assembler language, which is similar to machine code, except for the use of explicit labels rather than addresses as

destinations for jumps.

```
 ---------------  compil  ---------
| sstatement, |----->----| acode |
| address     |           ---------
 ---------------

 -----------  mcodetop  ---------
| acode,   |------>------| mcode |
| address  |              ---------
 -----------
```

The source statement to assembler phase itself splits into two distinct phases. Firstly allocate space for the variables declared at the head of blocks and replace all mention of them by their corresponding address. This is done by pstmttop, producing psstatements. Psstatements are identical to sstatements except for the use of addresses in place of variables and the omission of blocks (since blocks served only to declare variables).

```
 ---------------  pstmttop  ---------------
| sstatement, |------>-----| psstatement |
| address     |             ---------------
 ---------------
```

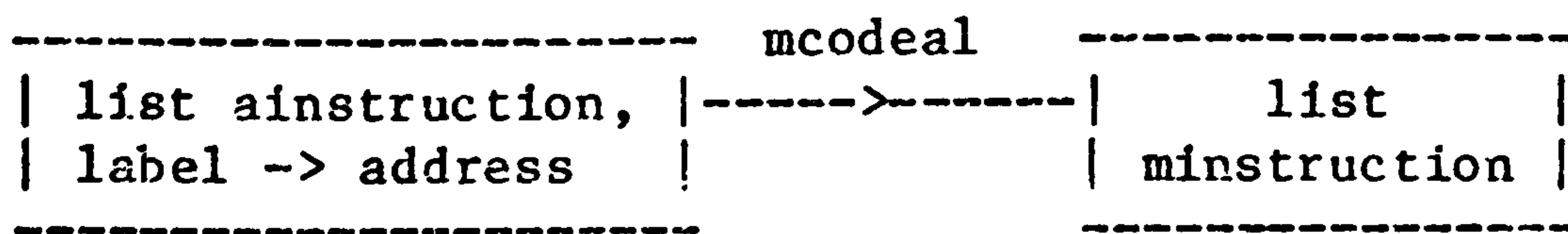
Secondly, convert the psstatements into assembler. This is done by introducing the appropriate assembler instructions, together with labels and jumps where required, for each construct of the sstatements.

```
 ---------------  acodetop  ---------
| psstatement |------>------| acode |
 ---------------             ---------
```

Converting assembler code to machine code is simply a matter of determining locations for instructions, and replacing jumps to labels by jumps to the appropriate location.

This conversion is achieved by first constructing a map from labels to addresses. Once such a map has been constructed, a simple pass through will replace all occurrences of labels by their

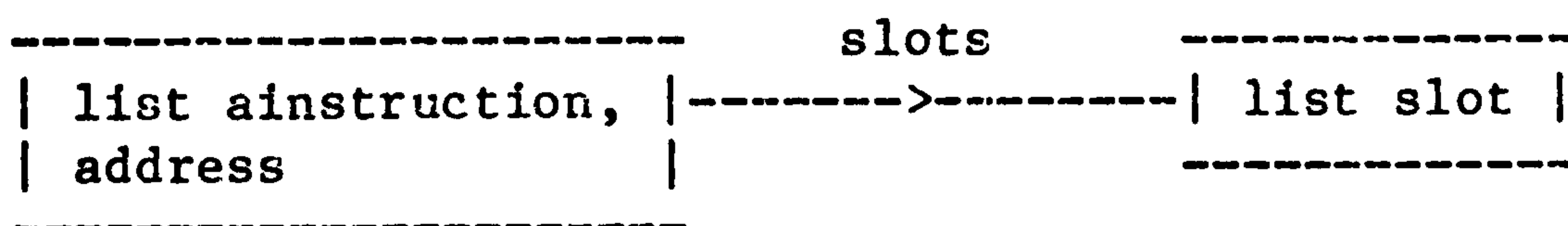
corresponding address - this pass is done by function mcodeal.

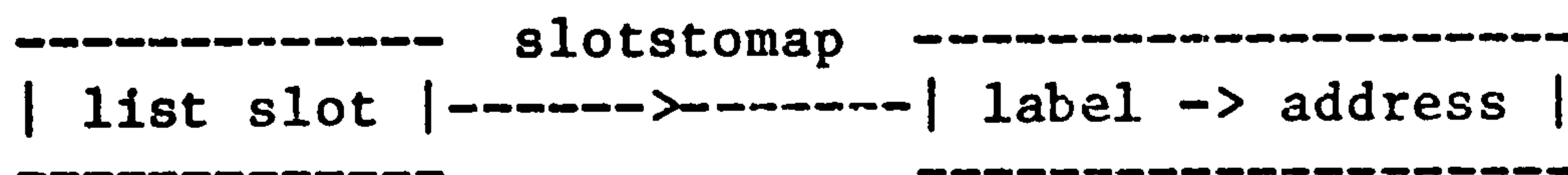


To construct the map, we first place consecutive assembler instructions into "slots", where a slot is a data structure consisting of an address and an assembler instruction.

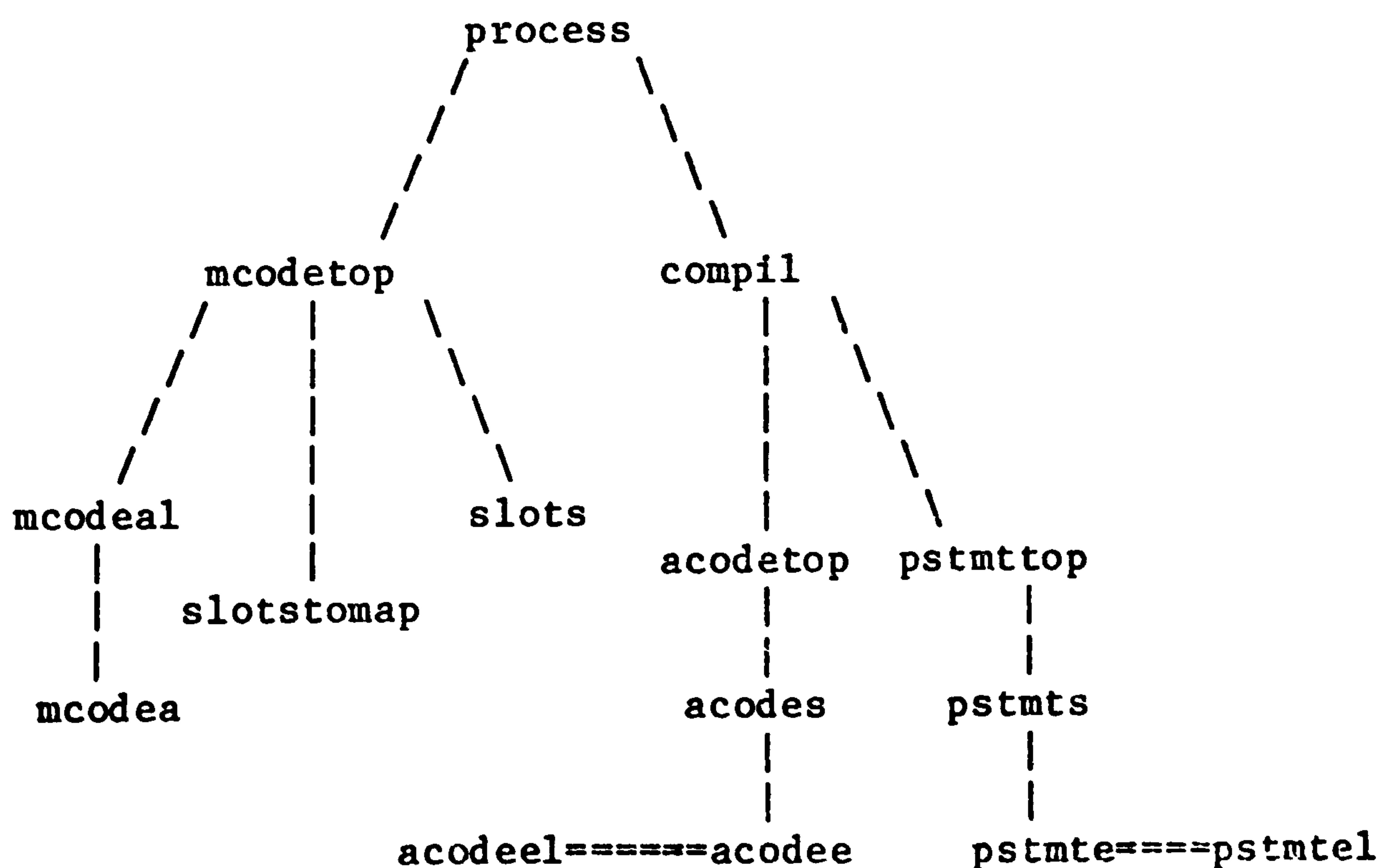
DATA slot <= sl(address,ainstruction)

From these a pass through the slots builds a map from the labels on instructions to their corresponding addresses.





The structure of the protoprogram is as follows (in the diagram "="'s between two functions indicates they are mutually recursive):



5.4.2 NPL Protoprogram

```
COMMENT '   length defined for lists @;
DEF
VAR A : atom   VAR L : list atom
+++ length(list atom) <= num
--- length(nil) <= 0
--- length(A::L) <= succ length(L)
END


COMMENT '   equality for lists @;
DEF
VAR A,A1 : atom   VAR L,L1 : list atom
+++ list atom = list atom <= truval
--- nil=nil <= true
--- nil=A1::L1 <= false
--- A::L=nil <= false
--- A::L=A1::L1 <= cond(A=A1,L=L1,false)
END


COMMENT '   maps and associated operations.
            nilfm is empty map.  mplus to update/add to a map.
            of to lookup. @;
DEF
INF 4 ->   INF 4 mplus   INF 4 of
DATA ALFA->BETA <= into(list(tuple2(ALFA,BETA)))

VAR AF1,AF2 : ALFA   VAR BE1,BE2 : BETA   VAR AFBEL : list
 (tuple2(ALFA,BETA))

+++ nilfm <= GENERAL->GENERAL
--- nilfm <= into(nil)

+++ (tuple2(ALFA,BETA)) mplus (ALFA -> BETA) <= (ALFA -> BETA)
--- <AF1,BE1> mplus into(AFBEL) <= into(<AF1,BE1> ::AFBEL)

+++ (ALFA -> BETA) of ALFA <= BETA
--- into(<AF1,BE1> ::AFBEL) of AF2 <= cond(AF1=AF2,
                                          BE1,
                                          into(AFBEL) of AF2)
END


COMMENT '   addresses, with incadd to generate next address @;
DEF     VAR N : num
DATA address <= ad(num)
VAR ADR,ADR1,CODEADR,VARADR : address
+++ incadd(address) <= address
--- incadd(ad(N)) <= ad(succ N)
END
```

```
DEF
DATA variable <= varible
VAR V,V1 : variable
VAR VL,VL1 : list variable
END


COMMENT '   environment and associated operations.
            An environment is a map from variables to addresses,
            together with the next free address.
            lookup for looking up the address of a variable.
            addtoenv for updating and environment when have a list
            of variables to allocate space for.
            nilenv is empty environment. @;
DEF     VAR VAMAP : variable -> address
DATA environment <= en(variable->address,address)
VAR ENV,ENV1 : environment
+++ nilenv(address) <= environment
--- nilenv(ADR) <= en(nilfm,ADR)
+++ addtoenv(list variable,environment) <= environment
+++ lookup(variable,environment) <= address
--- addtoenv(nil,ENV) <= ENV
--- addtoenv(V::VL,en(VAMAP,ADR)) <= addtoenv(VL,
                                     en(<V,ADR> mplus VAMAP,
                                        incadd(ADR) ) )
--- lookup(V,en(VAMAP,ADR)) <= VAMAP of V
END


COMMENT '   sstatement (source statement) and psstatement - which is
            an sstatement modified by replacing variables with
            addresses @;
DEF
DATA opr <= ops
VAR OP,OP1 : opr

INF 3 $
DATA sstatement <= asst(variable,expression)
                ++ whst(expression,sstatement)
                ++ ifthenelse(expression,sstatement,sstatement)
                ++ bl(list variable,sstatement)
                ++ sstatement $ sstatement ;
     expression <= expr(variable) ++ apply(opr,list expression)
VAR S,S1,S2 : sstatement
VAR E,E1 : expression
VAR EL,EL1 : list expression

DATA psstatement <= passt(address,pexpression)
                 ++ pwhst(pexpression,psstatement)
                 ++ pifthenelse(pexpression,psstatement,psstatement)
                 ++ psstatement $ psstatement ;
     pexpression <= pexpr(address) ++ papply(opr,list pexpression)
VAR PS,PS1,PS2 : psstatement
VAR PE,PE1 : pexpression
VAR PEL,PEL1 : list pexpression
END
```

```
COMMENT '   pstmttop replaces variables by addresses, allocating
            space for variables from and including given address. @;
DEF
+++ pstmttop(sstatement,address) <= psstatement

+++ pstmts(sstatement,environment) <= psstatement

+++ pstmte(expression,environment) <= pexpression

+++ pstmtel(list expression,environment) <= list pexpression


--- pstmttop(S,ADR) <= pstmts(S,nilenv(ADR))

--- pstmts(bl(VL,S),ENV) <= pstmts(S,addtoenv(VL,ENV))
--- pstmts(S1$S2,ENV) <= pstmts(S1,ENV) $ pstmts(S2,ENV)
--- pstmts(asst(V,E),ENV) <= passt(lookup(V,ENV),pstmte(E,ENV))
--- pstmts(whst(E,S),ENV) <= pwhst(pstmte(E,ENV),pstmts(S,ENV))
--- pstmts(ifthenelse(E,S1,S2),ENV) <= pifthenelse(pstmte(E,ENV),
                                      pstmts(S1,ENV),pstmts(S2,ENV))

--- pstmte(expr(V),ENV) <= pexpr(lookup(V,ENV))
--- pstmte(apply(OP,EL),ENV) <= papply(OP,pstmtel(EL,ENV))

--- pstmtel(nil,ENV) <= nil
--- pstmtel(E::EL,ENV) <= pstmte(E,ENV)::pstmtel(EL,ENV)
END


COMMENT '   generating unique labels @;
DEF     VAR NL : list num
DATA label <= lbl(list num)
VAR L1,L2,L3,L4,LAB : label

+++ nillab <= label
--- nillab <= lbl(nil)

+++ newlab2(label) <= TUPLE2(label,label)
--- newlab2(lbl(NL)) <= < lbl(0::NL),lbl(1::NL) >

+++ newlab3(label) <= TUPLE3(label,label,label)
--- newlab3(lbl(NL)) <= < lbl(0::NL),lbl(1::NL),lbl(2::NL) >

+++ newlab4(label) <= TUPLE4(label,label,label,label)
--- newlab4(lbl(NL)) <= < lbl(0::NL),lbl(1::NL),
                          lbl(2::NL),lbl(3::NL) >
END


COMMENT '   assembler code @;
DEF
DATA ainstruction <= load(address) ++ str(address) ++ condjump(label)
                  ++ jump(label) ++ app(opr) ++ nonop
                  ++ labins(label,ainstruction)
VAR AI,AI1 : ainstruction
```

```
VAR AIL,AIL1 : list ainstruction
DATA acode <= apr(list ainstruction)
VAR ACO : acode
END


COMMENT '   acodetop converts source code with addresses into
            assembler putting in labels where needed @;
DEF
+++ acodetop(psstatement) <= acode

+++ acodes(psstatement,label) <= list ainstruction

+++ acodee(pexpression) <= list ainstruction

+++ acodeel(list pexpression) <= list ainstruction

+++ insertlabel(label,list ainstruction) <= list ainstruction


--- acodetop(PS) <= apr(acodes(PS,nillab))

--- acodes(PS1$PS2,LAB) <= acodes(PS1,L1) <> acodes(PS2,L2)
                              where <L1,L2> == newlab2(LAB)

--- acodes(passt(ADR,PE),LAB) <= acodee(PE)<>[str(ADR)]

--- acodes(pwhst(PE,PS),LAB) <= insertlabel(L1,acodee(PE))
                          <>[condjump(L2)]
                          <>acodes(PS,L3)
                          <>[jump(L1)]
                          <>insertlabel(L2,nil)
                         where <L1,L2,L3> == newlab3(LAB)

--- acodes(pifthenelse(PE,PS1,PS2),LAB) <= acodee(PE)
                                  <>[condjump(L1)]
                                  <>acodes(PS2,L3)
                                  <>[jump(L2)]
                              <>insertlabel(L1,acodes(PS1,L4))
                              <>insertlabel(L2,nil)
                           where <L1,L2,L3,L4> == newlab4(LAB)

--- acodee(pexpr(ADR)) <= [load(ADR)]
--- acodee(papply(OP,PEL)) <= acodeel(PEL)<>[app(OP)]

--- acodeel(nil) <= nil
--- acodeel(PE::PEL) <= acodee(PE) <> acodeel(PEL)

--- insertlabel(LAB,AIL) <= labins(LAB,nonop)::AIL
END


COMMENT '   compil converts source code to assembler, allocating
            space for variables starting from given address. @;
DEF
```

```
+++ compil(sstatement,address) <= acode
--- compil(S,VARADR) <= acodetop(pstmttop(S,VARADR))
END



COMMENT '   slots are pairs of address-assembler instruction @;
DEF
DATA slot <= sl(address,ainstruction)
VAR SLTL : list slot
END

COMMENT '   slots makes a list of slots from assembler code,
            putting code in slots with successive addresses,
            starting at given address.
            slotstomap builds a map from labels to addresses. @;
DEF
+++ slots(list ainstruction,address) <= list slot
+++ slotstomap(list slot) <= label -> address

--- slots(nil,ADR) <= nil
--- slots(AI::AIL,ADR) <= sl(ADR,AI) ::
                               slots(AIL,incadd(ADR))

--- slotstomap(nil) <= nilfm
--- slotstomap(sl(ADR,load(ADR1))::SLTL) <=
                          slotstomap(SLTL)
--- slotstomap(sl(ADR,str(ADR1))::SLTL) <= slotstomap(SLTL)
--- slotstomap(sl(ADR,condjump(LAB))::SLTL) <=
                           slotstomap(SLTL)
--- slotstomap(sl(ADR,app(OP))::SLTL) <= slotstomap(SLTL)
--- slotstomap(sl(ADR,jump(LAB))::SLTL) <= slotstomap(SLTL)
--- slotstomap(sl(ADR,nonop)::SLTL) <= slotstomap(SLTL)
--- slotstomap(sl(ADR,labins(LAB,AI))::SLTL) <=
                     <LAB,ADR> mplus slotstomap(SLTL)
END



COMMENT '   machine code @;
DEF
DATA minstruction <= mload(address) ++ mstr(address)
                     ++ mcondjump(address)  ++ mjump(address)
                     ++ mapp(opr) ++ mnonop
DATA mcode <= mpr(list minstruction)
END

COMMENT '   mcodetop converts assembler to machine code @;
DEF
VAR labaddmap : label->address

+++ mcodetop(acode,address) <= mcode
+++ mcodeal(list ainstruction,label->address) <= list minstruction
+++ mcodea(ainstruction,label->address) <= minstruction
```

```
--- mcodetop(apr(AIL),CODEADR) <=
     mpr(mcodeal(AIL,slotstomap(slots(AIL,CODEADR))))

--- mcodeal(nil,labaddmap) <= nil
--- mcodeal(AI::AIL,labaddmap) <= mcodea(AI,labaddmap)
                                  ::mcodeal(AIL,labaddmap)
--- mcodea(labins(LAB,AI),labaddmap) <= mcodea(AI,labaddmap)
--- mcodea(load(ADR),labaddmap) <= mload(ADR)
--- mcodea(str(ADR),labaddmap) <= mstr(ADR)
--- mcodea(condjump(LAB),labaddmap) <= mcondjump(labaddmap of LAB)
--- mcodea(jump(LAB),labaddmap) <= mjump(labaddmap of LAB)
--- mcodea(app(OP),labaddmap) <= mapp(OP)
--- mcodea(nonop,labaddmap) <= mnonop
END


COMMENT '   process converts source code to machine code, allocating
            variable space from and including VARADR, allocating code
            space from and including CODEADR. @;
DEF
+++ process(sstatement,address,address) <= mcode
--- process(S,CODEADR,VARADR) <= mcodetop(compil(S,VARADR),CODEADR)
END
```

5.4.3 Transforming To Efficient Version

Adopting the simple strategy outlined in section 5.2 suggests we tackle the transformation of the above protoprogram in the following stages:

(i) Improve compil by

Combining acodels(S,ENV,LAB) <= acodes(pstmts(S,ENV),LAB)

```
                        process
                       /       \
                      /         \
                     /           \
                    /             \
           mcodetop                compil
          /    |    \                |
         /     |     \               |
        /      |      \            acodels
       /       |       \             |
mcodeal        |        slots        |
   |           |                   acodele===acodelel
   |       slotstomap
   |
mcodea
```

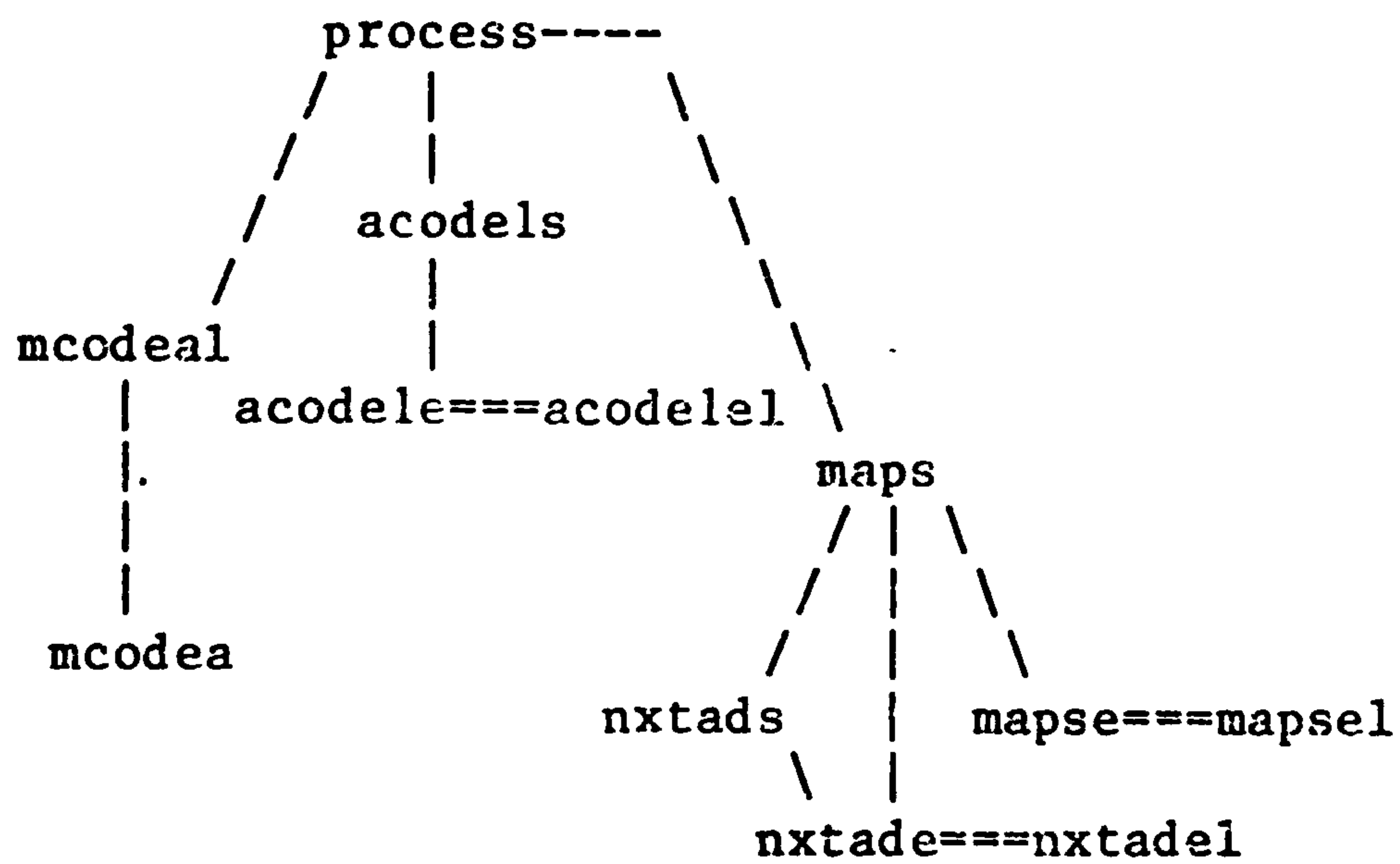
(ii) Improve mcodetop by

Combining map(AIL,ADR) <= slotstomap(slots(AIL,ADR))

```
                        process
                       /       \
                      /         \
                     /           \
                    /             \
          mcodetop              compil
          /      |                |
         /       |                |
        /        |              acodels
       /         |                |
mcodeal         map               |
   |                            acodele===acodelel
   |
mcodea
```

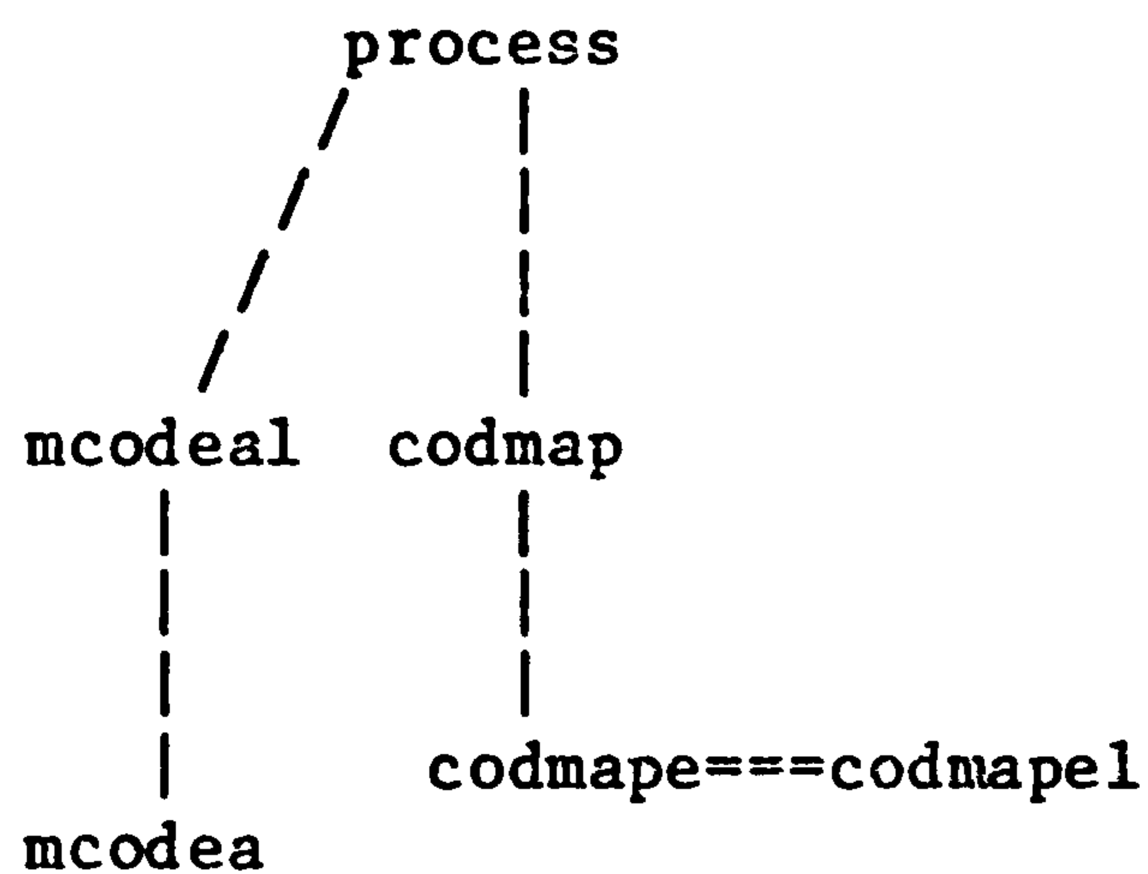
(iii) Improve process by

Combining maps(S,ENV,LAB,ADR) <=
map(acodels(S,ENV,LAB),ADR)



(iv) Improve process by

Tupling maps and acodels together



Now I present the commands for each stage as given to the ZAP system to carry out the transformations:

(i) Combining acodes and pstmts

```
CONTEXT
   UNFOLDALL compil
   USING nilenv nillab
         RESTRICTED acodes pstmts
   TRANSFORM
```

```
      GOAL compil(S,ADR) <= $$(&&acodels(S,nilenv(ADR),nillab))
   END
END

CONTEXT
   UNFOLD acodels acodes acodee acodeel pstmts pstmte pstmtel
          nilenv addtoenv insertlabel
   USING <> addtoenv lookup newlab2 newlab3 newlab4
        RESTRICTED acodels acodes pstmts
                   acodee acodeel pstmte pstmtel

   TRANSFORM
      GOAL acodels(asst(V,E),ENV,LAB) <= $$(&&acodele(ENV,E),V,ENV)
      GOAL acodels(whst(E,S),ENV,LAB) <=
                  $$(acodele(ENV,E),acodels(S,ENV,$$(LAB)),ENV,LAB)
      GOAL acodels(ifthenelse(E,S1,S2),ENV,LAB) <=
                              $$(acodele(ENV,E),
                                 acodels(S1,ENV,$$(LAB)),
                                 acodels(S2,ENV,$$(LAB)),
                                 ENV,LAB)
      GOAL acodels(bl(VL,S),ENV,LAB) <= acodels(S,$$(VL,ENV),LAB)
      GOAL acodels(S1$S2,ENV,LAB) <= $$(acodels(S1,ENV,$$(LAB)),
                                  acodels(S2,ENV,$$(LAB)))
      GOAL acodele(ENV,expr(V))
      GOAL acodele(ENV,apply(OP,EL)) <= $$(&&acodelel(ENV,EL),OP)
      GOAL acodelel(ENV,nil)
      GOAL acodelel(ENV,E::EL) <= $$(acodele(ENV,E),acodelel(ENV,EL))
   END

   DELETE acodels(S,ENV,LAB),
          acodele(ENV,E),
          acodelel(ENV,EL)

END
```

Note that acodels is extended to acodele and acodelel in order to handle expressions and lists of expressions respectively; this mirrors the original structure of pstmts/pstmte/pstmtel.

(ii) Combining slotstomap and slots

```
CONTEXT
   UNFOLDALL mcodetop
   USING mcodeal
        RESTRICTED slotstomaplots
   TRANSFORM
      GOAL mcodetop(apr(AIL),CODEADR) <= $$(AIL,&&map(AIL,CODEADR))
   END
END

CONTEXT
   UNFOLDALL map
```

```
   USING incadd mplus nilfm
        RESTRICTED map
   TRANSFORM
      GOAL map(nil,ADR) <= nilfm
      GOAL map((CASESOF AI)::AIL,ADR) <= map(AIL,$$(ADR))
      GOAL map(labins(LAB,AI)::AIL,ADR) <=
 $$(LAB,ADR,map(AIL,$$(ADR)))
   END
END
```

Note that stages (i) and (ii) are completely independant, and may be performed in either order.

(iii) Combining map and acodels

This stage requires some insight into the processes involved. Attempting the transformations of the combination, maps, leads to considering ga applied to lists of ainstructions appended together, e.g. map(AIL1<>AIL2,ADR). Because map is defined in terms of cases of the head of a list of ainstructions (if non-null), nothing can be done with such an expression.

We know, however, that map is simply constructing a label->address map for the ainstructions placed in successive addresses commencing at ADR. So, for an appended pair of lists AIL1<>AIL2, the result will be map(AIL1,ADR) with map(AIL2,ADR′) added on, where ADR′ is the next free address after instructions of AIL1 have been allocated space. Hence we are motivated to introduce a function to compute this next free address, nxtad, and then we have

```
map(AIL1<>AIL2,ADR) <= map(AIL2,nxtad(ADR,AIL1)) addmaps
                                 map(AIL1,ADR)
```

nxtad has an easy definition, and also satisfies

```
nxtad(incadd(ADR),AIL) <= incadd(nxtad(ADR,AIL))

nxtad(ADR,AIL1<>AIL2) <= nxtad(nxtad(ADR,AIL1),AIL2)
```

The transformation commands are:

```
CONTEXT
   UNFOLD process mcodetop nillab compil nilenv
   USING mcodeal nilenv nillab
         RESTRICTED acodels map
   TRANSFORM
      GOAL process(S,CODEADR,VARADR) <=
                   mpr(mcodeal(acodels(S,nilenv(VARADR),nillab),
                   &&maps(S,nilenv(VARADR),nillab,CODEADR)))
   END
END


INTRODUCE
VAR ADR2 : address    VAR AIL2 : list ainstruction
VAR AFBEL1 : list tuple2(ALFA,BETA)
INF 4 addmaps
+++ (ALFA->BETA) addmaps (ALFA->BETA) <= (ALFA->BETA)

--- into(NIL) addmaps into(AFBEL1) <= into(AFBEL1)
--- into(<AF1,BE1> ::AFBEL) addmaps into(AFBEL1) <=
         <AF1,BE1> mplus (into(AFBEL) addmaps into(AFBEL1))

+++ nxtad(ADDRESS,LIST AINSTRUCTION) <= ADDRESS
--- nxtad(ADR,NIL) <= ADR
--- nxtad(ADR,AI::AIL) <= incadd(nxtad(ADR,AIL))
END

CONTEXT
   USING addtoenv newlab2 newlab3 newlab4 <> lookup incadd addmaps
         mplus nilfm
         RESTRICTED maps map acodele acodelel nxtad acodels
   UNFOLD maps acodels map nxtad addmaps acodele acodelel
   LEMMAS IDENTITY addmaps nilfm
   LEMMAS --- nxtad(incadd(ADR),AIL) <= incadd(nxtad(ADR,AIL))
          --- nxtad(ADR,AIL1<>AIL2) <= nxtad(nxtad(ADR,AIL1),AIL2)
          --- map(AIL1<>AIL2,ADR) <= map(AIL2,nxtad(ADR,AIL1))
                                addmaps map(AIL1,ADR)

   TRANSFORM

      GOAL maps(asst(V,E),ENV,LAB,ADR) <= &&mapse(E,ENV,ADR)

      GOAL maps(bl(VL,S),ENV,LAB,ADR) <= maps(S,$$(VL,ENV),LAB,ADR)

      GOAL maps(S1 $ S2,ENV,LAB,ADR) <=
                         $$(maps(S1,ENV,$$(LAB),ADR),
                      maps(S2,ENV,$$(LAB),&&nxtads(ADR,S1,ENV,L1)))
                where <L1,L2> == newlab2(LAB)

      GOAL maps(whst(E,S),ENV,LAB,ADR) <=
                              $$(LAB,ADR,nxtads(ADR1,S,ENV,$$(LAB)),
                                 maps(S,ENV,$$(LAB),$$(ADR1)),
                                 mapse(E,ENV,$$(ADR)))
                where <ADR1> == < &&nxtade(ADR,E,ENV) >
```

```
    GOAL maps(ifthenelse(E,S1,S2),ENV,LAB,ADR) <=
             $$(LAB,nxtads(ADR1,S1,ENV,$$(LAB)),ADR1,
                maps(S1,ENV,$$(LAB),$$(ADR1)),
                maps(S2,ENV,$$(LAB),$$(ADR2)),
                mapse(E,ENV,ADR))
          where <ADR1> == <nxtads(ADR2,S2,ENV,$$(LAB))>
          where <ADR2> == <nxtade(ADR,E,ENV)>


    GOAL mapse(expr(V),ENV,ADR)

    GOAL mapse(apply(OP,EL),ENV,ADR) <= &&mapsel(EL,ENV,ADR)


    GOAL mapsel(nil,ENV,ADR)

    GOAL mapsel(E::EL,ENV,ADR) <=
          $$(mapsel(EL,ENV,nxtade(ADR,E,ENV)),mapse(E,ENV,ADR))


    GOAL nxtads(ADR,S1 $ S2,ENV,LAB) <=
              nxtads(nxtads(ADR,S1,ENV,$$(LAB)),S2,ENV,$$(LAB))

    GOAL nxtads(ADR,bl(VL,S),ENV,LAB) <=
                        nxtads(ADR,S,$$(VL,ENV),LAB)

    GOAL nxtads(ADR,asst(V,E),ENV,LAB) <= $$(nxtade(ADR,E,ENV))

    GOAL nxtads(ADR,whst(E,S),ENV,LAB) <=
                   $$(nxtads(nxtade(ADR,E,ENV),S,ENV,$$(LAB)))

    GOAL nxtads(ADR,ifthenelse(E,S1,S2),ENV,LAB) <=
           $$(nxtads(nxtads(nxtade(ADR,E,ENV),
                     S2,ENV,$$(LAB)),S1,ENV,$$(LAB)))


    GOAL nxtade(ADR,expr(V),ENV)

    GOAL nxtade(ADR,apply(OP,EL),ENV) <= $$(&&nxtadel(ADR,EL,ENV))


    GOAL nxtadel(ADR,nil,ENV)

    GOAL nxtadel(ADR,E::EL,ENV) <=
                       nxtadel(nxtade(ADR,E,ENV),EL,ENV)

  END

END

DELETE
  maps(S,ENV,LAB,ADR),
  mapse(E,ENV,ADR),
  mapsel(EL,ENV,ADR),
  nxtads(ADR,S,ENV,LAB),
  nxtade(ADR,E,ENV),
```

nxtadel(ADR,EL,ENV)

Note that nxtads is essentially the combination of nxtad with acodels. Again, as in stage (i), the new functions extend to cover expressions and lists of expressions.

(iv) Tuple codmap(S,ENV,LAB,ADR) <=

< acodels(S,ENV,LAB),maps(S,ENV,LAB,ADR),nxtads(ADR,S,ENV,LAB) >

Our original plan was to tuple together acodelsand maps, but by this stage we have introduced nxtads, which can also profitably be computed at the same time.

Once again we introduce functions codmape and codmapel to handle expressions and lists of expressions.

```
/// 'introduce functions to tuple together h, gah and nh @
INTRODUCE
   VAR T1,T2,T3,T1A,T2A,T3A,T1B,T2B,T3B : general

   +++ codmap(sstatement,environment,label,address) <=
             tuple3(list ainstruction,label -> address,address)
   --- codmap(S,ENV,LAB,ADR) <= <acodels(S,ENV,LAB),
                 maps(S,ENV,LAB,ADR),nxtads(ADR,S,ENV,LAB)>

   +++ codmape(expression,environment,address) <=
             tuple3(list ainstruction,label -> address,address)
   --- codmape(E,ENV,ADR) <= <acodele(ENV,E),
                     mapse(E,ENV,ADR),nxtade(ADR,E,ENV)>

   +++ codmapel(list expression,environment,address) <=
             tuple3(list ainstruction,label -> address,address)
   --- codmapel(EL,ENV,ADR) <= <acodelel(ENV,EL),
                     mapsel(EL,ENV,ADR),nxtadel(ADR,EL,ENV)>

END


CONTEXT
   USING mcodeal nilenv codmap
   UNFOLD process codmap
   TRANSFORM
      GOAL process(S,CODEADR,VARADR) <=
                     $$(codmap(S,nilenv(VARADR),nillab,CODEADR))
   END
END
```

```
CONTEXT
   USING codmap codmape codmapel    incadd <> mplus addmaps
         newlab2 newlab3 newlab4 addtoenv lookup
   UNFOLD codmap codmape codmapel acodels maps nxtads
   LEMMAS --- nxtads(incadd(ADR),S,ENV,LAB) <=
                                  incadd(nxtads(ADR,S,ENV,LAB))
          --- nxtade(incadd(ADR),E,ENV) <=
                                  incadd(nxtade(ADR,E,ENV))
          --- nxtadel(incadd(ADR),EL,ENV) <=
                                  incadd(nxtadel(ADR,EL,ENV))

   TRANSFORM
      GOAL codmap(S1 $ S2,ENV,LAB,ADR) <= $$(T1,T2,T3,T1A,T2A,T3A)
          where <T1A,T2A,T3A> == codmap(S2,ENV,$$(LAB),$$(T1,T2,T3))
              where <T1,T2,T3> == codmap(S1,ENV,$$(LAB),ADR)

      GOAL codmap(bl(VL,S),ENV,LAB,ADR) <=
                               codmap(S,$$(ENV,VL),LAB,ADR)

      GOAL codmap(asst(V,E),ENV,LAB,ADR) <=
                              $$(codmape(E,ENV,ADR),V,ENV,LAB)

      GOAL codmap(whst(E,S),ENV,LAB,ADR) <=
              $$(LAB,ENV,ADR,T1,T2,T3,T1A,T2A,T3A)
              where <T1A,T2A,T3A> == codmap(S,ENV,$$(LAB),incadd(T3))
                 where <T1,T2,T3> == codmape(E,ENV,incadd(ADR))

      GOAL codmap(ifthenelse(E,S1,S2),ENV,LAB,ADR) <=
                  $$(ENV,LAB,ADR,T1,T2,T1A,T2A,T3A,T1B,T2B,T3B)
           where <T1B,T2B,T3B> ==
                        codmap(S1,ENV,$$(LAB),incadd(incadd(T3A)))
           where <T1A,T2A,T3A> == codmap(S2,ENV,$$(LAB),incadd(T3))
              where <T1,T2,T3> == codmape(E,ENV,ADR)
   END

   CONTEXT
      UNFOLD acodele mapse nxtade acodelel mapsel nxtadel
      TRANSFORM
         GOAL codmape(expr(V),ENV,ADR)

         GOAL codmape(apply(OP,EL),ENV,ADR) <=
                    $$(OP,codmapel(EL,ENV,ADR))


         GOAL codmapel(nil,ENV,ADR)

         GOAL codmapel(E::EL,ENV,ADR) <= $$(T1,T2,T3,T1A,T2A,T3A)
               where <T1A,T2A,T3A> == codmapel(EL,ENV,$$(T1,T2,T3))
               where <T1,T2,T3> == codmape(E,ENV,ADR)
      END
   END



   DELETE
      codmap(S,ENV,LAB,ADR), codmape(E,ENV,ADR), codmapel(EL,ENV,ADR)
```

END

5.4.4 Final Program

The final program is a two pass compiler, the first pass being used to create assembler code with labels, and a map from labels to addresses. The second pass takes these and replaces each label in the code by its corresponding address (looked up in the map) to get the final machine code.

The change from the protoprogram - a very naive solution, with many passes through the input - has again been very major. The functions achieve the efficiency by intertwining several actions at once, returning several results.

```
--- codmapel(E :: EL,ENV,ADR) <= <U500 <> U503,U502 addmaps
                                         U499,U501>
                 where <U503,U502,U501> ==  codmapel(EL,ENV,U498)
                 where <U500,U499,U498> ==  codmape(E,ENV,ADR)

--- codmapel(nil,ENV,ADR) <= <nil,nilfm,ADR>

--- codmape(apply(OP,EL),ENV,ADR) <=
                   <U464 <> app(OP) :: nil,U463,incadd(U462)>
           where <U464,U463,U462> ==  codmapel(EL,ENV,ADR)

--- codmape(expr(V),ENV,ADR) <=
                   <load(lookup(V,ENV)) :: nil,nilfm,incadd(ADR)>

--- codmap(S1 $ S2,ENV,LAB,ADR) <= <U110 <> U113,
                                      U112 addmaps U109,U111>
                where <U113,U112,U111> ==  codmap(S2,ENV,U106,U108)
                  where <U110,U109,U108> ==  codmap(S1,ENV,U107,ADR)
                   where <U107,U106> ==  newlab2(LAB)

--- codmap(bl(VL,S),ENV,LAB,ADR) <=
                         codmap(S,addtoenv(VL,ENV),LAB,ADR)

--- codmap(asst(V,E),ENV,LAB,ADR) <=
                <U181 <> STR(lookup(V,ENV)) :: nil,U180,incadd(U179)>
                      where <U181,U180,U179> ==  codmape(E,ENV,ADR)

--- codmap(whst(E,S),ENV,LAB,ADR) <=
                   <(((labins(U366,nonop) :: U369
                     <> condjump(U365) :: nil) <> U372)
```

```
                    <> jump(U366) :: nil)
                    <> labins(U365,nonop) :: nil ,
                   (<U365,incadd(U370)> mplus nilfm)
                    addmaps(U371 addmaps(<U366,ADR> mplus U368)),
                   incadd(incadd(U370))>
        where <U372,U371,U370> ==  codmap(S,ENV,U364,incadd(U367))
         where <U366,U365,U364> ==  newlab3(LAB)
          where <U369,U368,U367> ==  codmape(E,ENV,incadd(ADR))

--- codmap(ifthenelse(E,S1,S2),ENV,LAB,ADR) <=
                    <((((U400 <> condjump(U394) :: nil)
                      <> U397) <> jump(U393) :: nil)
                      <>labins(U394,nonop) :: U390)
                      <>labins(U393,nonop) :: nil ,
                     (<U393,U388> mplus nilfm)
                      addmaps ((<U394,incadd(U395)> mplus U389)
                      addmaps(U396 addmaps U399)) ,
                     incadd(U388) >
   where <U390,U389,U388> == codmap(S1,ENV,U391,incadd(incadd(U395)))
    where <U397,U396,U395> == codmap(S2,ENV,U392,incadd(U398))
     where <U400,U399,U398> == codmape(E1,ENV,ADR)
      where <U394,U393,U392,U391> == newlab4(LAB)

--- process(S,CODEADR,VARADR) <=  mpr(mcodeal(U6,U5))
        where <U6,U5,U4> ==  codmap(S,nilenv(VARADR),nillab,CODEADR)
```

(definitions of newlab and mcodeal remain unchanged)

5.5 REMARKS ON TRANSFORMATION EXAMPLES

Transformation of the telegram problem took place in parallel with, and influenced, my development of the transformation system. Hence it is impossible to give any meaningful estimate of the amount of effort it took to perform. At the time it seemed a hard transformation, although looking back on it in the light of further experience of transforming, I now consider it would be a reasonably straightforward problem to tackle. The compiler example served as a test of the system to ensure it was capable of transforming more than just the telegram problem. The system required no major modification; some aspects of its operation needed improvement and the command language was somewhat rationalised, but on the whole the system, and the transformation strategy, sufficed to complete the example. Some effort was required to design the simple protoprogram, notably in the need to make a concious decision to aim for clarity rather than efficiency (the habit of considering efficiency is hard to dispel). The transformation took place over several weeks, and again I feel I could do the transformation much more easily now having had more experience.

Both the programs are non-trivial when compared to the examples previous machine-based transformation work has been applied to. In comparison with "real" software, they are still quite small. Even so, an interesting feature is becoming apparent - the textual size of the protoprogram may be larger than the textual size of the transformed, efficient, program. For similarly sized examples we are able to keep in mind the entire program at once, so the advantages of modularity and clarity of the protoprograms are not very significant.

When moving to larger programs, for which we are unable to keep in mind all the details at once, modularity of the protoprogram becomes much more significant.

The next chapter presents the transformation of an example which is an order of magnitude larger than the examples here, where these issues become apparent.

CHAPTER 6

TRANSFORMATION OF A TEXT FORMATTER

In this chapter I present the transformation of a text formatter - a program for neatly formatting a document on a suitable printer. This example is considerably larger than those presented in the preceeding chapter, and its scale brings to light new problems during transformation. I discuss these problems and their implication for the practical development of programs by transformation.

Briefly, the difficulties which arise revolve around the difficulty of using the present tools and techniques to concisely capture simple changes taking place within large programs.

6.1 INFORMAL SPECIFICATION OF THE TEXT FORMATTER

The text formatter I construct and transform is based upon the formatter presented by Kernighan and Plauger in Chapter 7 of their book Software Tools (Kernighan and Plauger [1976]). In this book they present several pieces of software written in Ratfor, FORTRAN with extra syntax added. For each piece of software they show how it can be constructed in an organised fashion, but, as is common, their development goes straight from an informal specification to a final efficient program. I adopt their informal specification of a text formatter as the starting point for my development, and aim to end up

with a program of comparable efficiency to their hand-developed version, albeit in recursive form.

I repeat the informal specification given in Software Tools:

The format program described here is quite conventional. It accepts text to be formatted, interspersed with formatting commands telling the program what the output is to look like. A command consists of a period, a two-letter name, and perhaps some optional information. Each command must appear at the beginning of a line, with nothing on the line but the command and its arguments. For instance,

```
.ce
```

centres the next line of output, and

```
.sp 3
```

generates three blank lines.

Most of the time, however, the user should have to know little about commands and arguments - most formatting happens automatically. This is merely good human engineering. Ideally a document containing no commands should be printed sensibly. Default parameter settings and formatting actions are intended to be reasonable and free of suprises. For instance, words fill up output lines as much as possible, regardless of the length of input lines. Blank lines cause fresh paragraphs. Input is correctly spaced across page boundaries, with top and bottom margins.

At the same time the design has to be sufficiently flexible that it can be augmented with more advanced features for sophisticated use. Knowledgeable users should of course be able to change

parameter settings as desired. Ultimately it should be possible for users to define new formatting operations in terms of those already provided.

Commands

As we said, all commands are introduced by a period at the beginning of a line, which is an unlikely combination in text, and have two-letter names. It has been our experience that users prefer concise commands in most languages, so this seems a reasonable compromise between brevity and mnemonic value.

By default the program fills output lines, by packing as many input words as possible onto an output line before printing it. The lines are also justified (right margins made even) by inserting extra spaces into the filled line before output. People normally want filled text, which is why we choose it as the default behaviour. It can be turned off, however, by the no-fill command.

```
.nf
```

and thereafter lines will be copied from input to output without any rearrangement. Filling can be turned back on with the fill command

```
.fi
```

When an .nf is encountered, there may be a partial line collected but not yet output. The .nf will force this line out before anything else happens. The action of forcing out a partially collected line is called a break. The break concept pervades format; many commands implicitly cause a break. To force a break explicitly, for example to separate two paragraphs, use

```
.br
```

Of course you may want to add an extra blank line between paragraphs. The space command

```
.sp
```

causes a break, then produces a blank line. To get n blank lines, use

```
.sp n
```

(A space is always required between a command and its argument). If the bottom of a page is reached before all of the blank lines have been printed, the excess ones are thrown away, so that all pages will normally start at the same first line.

By default output will be single spaced, but line spacing can be changed at any time:

```
.ls n
```

sets the line spacing to n. (n=2 is double spacing) The .ls command does not cause a break.

The begin page command .bp causes a skip to the top of a new page and also causes a break. If you use

```
.bp n
```

the next output page will be numbered n. A .bp that occurs at the bottom of a page has no effect except perhaps to set the page number; no blank page is generated. The current page length can be changed (without a break) with

```
.pl n
```

To center the next line of output,

```
.ce
line to be centred
```

The .ce command causes a break. You can center n lines with

```
.ce n
```

and, if you don't like to count lines (or can't count correctly), say

```
.ce 1000
lots of lines
to be centred
.ce 0
```

The lines between the .ce commands will be centred. No filling is done on centred lines.

Underlining is much the same as centering:

```
.ul n
```

causes the text on the next n lines to be underlined upon output. But .ul does not cause a break, so words in filled text may be underlined by

```
words and words and
.ul
lots more
words.
```

to get

words and words and <u>lots</u> <u>more</u> words.

Centering and underlining may be intermixed in any order:

```
.ce
.ul
Title
```

gives a centred and underlined title.

The indent command controls the left margin:

```
.in n
```

causes all subsequent output lines to be indented n positions. (Normally they are indented by 0.) The command

```
.rm n
```

sets the right margin to n. The line length of filled lines is the difference between right margin and indent values. .in and .rm do not cause a break.

The traditional paragraph indent is produced with temporary

indent command:

```
.ti n
```

breaks and sets the indent relative to position n for one output line only. If n is less than the current indent, the indent is backwards (a "hanging indent").

To put running header and footer titles on every page, use .he and .fo:

```
.he this becomes the top-of-page (header) title
.fo this becomes the bottom-of-page (footer) title
```

The title begins with the first non-blank after the command, but a leading quote will be discarded if present, so you can produce titles that begin with blanks. If a title contains the character #, it will be replaced by the current page number each time the title is actually printed. .he and .fo do not cause a break.

Since absolute numbers are often awkward, the program allows relative values as command arguments. All commands that allow a numeric argument n also allow +n or -n instead, to signify a change in the current value. For instance,

```
.rm -10
.in +10
```

shrinks the right margin by 10 from its current value, and moves the indent 10 places further to the right. Thus

```
.rm 10
```

and

```
.rm +10
```

are quite different.

Relative values are particularly useful with .ti, to temporarily indent relative to the current indent:

```
.in +5
.ti +5
```

produces a left margin indented by 5, with the first line indented by a further 5.

And

```
.in +5
.ti -5
```

produces a hanging indent, as in a numbered paragraph:

1. Now is the time for all good people
   to come to the party.

A line that begins with blanks is a special case. If there is no text at all, the line causes a break and produces a number of blank lines equal to the current line spacing. These lines are never discarded regardless of where they appear, so they provide a way to get blank lines to the top of a page. If a line begins with n blanks followed by text, it causes a break and a temporary indent of +n. These special actions help ensure that a document that contains no formatting commands will still be reasonably formatted.

In summary, then, we have the following commands. If a numeric argument is preceeded by a + or a -, the previous value is changed by this amount; otherwise the argument represents the new value. If no argument is given, the default value is used.

command	break ?	default	function
.bp n	yes	n=+1	begin page numbered n
.br	yes		cause break
.ce n	yes	n=1	center next n lines
.fi	yes		start filling
.fo	no	empty	footer title
.he	no	empty	header title
.in n	no	n=0	indent n spaces
.ls n	no	n=1	line spacing is n
.nf	yes		stop filling
.pl n	no	n=66	set page length to n
.rm n	no	n=60	set right margin to n
.sp n	yes	n=1	space down n lines
.ti n	yes	n=0	temporary indent of n
.ul n	no	n=1	underline words from next n lines

6.2 DESIGN OF PROTOPROGRAM

A major difficulty in describing text formatting is that it admits to no concise yet precise specification. In practice this leads to user manuals which are verbose yet unable to answer all the questions the user wants to ask. Users typically resort to a "try it and see" approach in order to determine the behaviour of their formatter, gradually accumulating a set of tricks to enable them to achieve the effects they desire.

In contrast I have attempted to discard the mantle of efficiency, and design as clear a program as possible. I split the task of formatting into several stages which are as independent as possible, and reflect what I consider to be the conceptual stages themselves of the process.

These are:

(1) Decoding commands from text

(2) processing commands which can be used immediately to associate information with input text lines.

(3) Preliminary processing of text lines.

(4) Formation of lines for output.

(5) Formation of pages for output.

A detailed description of each stage follows:

Stage 1: - Decoding commands from text

In this stage the input lines which are commands are distinguished from those which are text lines, and interpreted. Some commands have an argument, which may be a string (as for HE and FO commands), or a number, either signed or unsigned. An illegal argument (e.g. a non-digit encountered when reading in a number) has the same effect as if no argument had been supplied.

Output of this stage is list torc (mnemonic for text or command), where torc has been defined by

DATA torc <= text(list char) ++ command $ carg

i.e. a text line, or command and its argument.

Stage 2: - Processing commands which associate information with text lines.

Many of the commands of a text formatter are involved with the setting of some value, which will have an effect of the output of the following text lines. For example, after a .rm 54 command, output lines will be fit within a right margin size of 54.

Suppose we consider each text line as consisting not only of a list of characters to be processed for output, but in addition associate with it information - such as the right margin size. Then we could regard many of the formatter commands as merely setting

certain values in this information in the following text lines, and how the information is used is left to later stages.

This is the approach I use in my program. The pieces of information set by the commands are stored in a data structure I call an infomap. Functions to add to and look up values in infomaps are simply defined. Each text line is modified to have an infomap associated with it.

The different types of values stored in the infomaps are:

```
PLVAL - page length
CEVAL - whether or not to centre this line
ULVAL - whether or not to underline this line
FIVAL - whether or not to fill this line
LSVAL - line spacing
HEVAL - page header title
FOVAL - page footer title
RMVAL - right margin
TIVAL - temporary indentation,
          applies to first output line generated
          from this input text line
INVAL - indentation, applies to later output lines
```

The TIVAL and INVAL values are a little less obvious than the rest: INVAL is the normal indentation, set by the in command. However, the effect of a .ti n command is to cause a break and start the next line with a temporary indentation of n. The need for both the temporary and normal values to be remembered arises from the use of "fill" mode of input. When in this mode, a single input text line may give rise to more than one output line, and in such a case, after a .ti n command, we want the first output line to have the temporary indent of n, but following lines to revert to the normal indentation. Hence the need to keep both values. Thus the first text line after each ti command will have the TIVAL indicating the temporary indent set by that command, other text lines will have their TIVAL's set to

whatever the normal indentation is, i.e. the same as their INVAL's.

With one exception, for each type of command there is a separate pass through the input to set its associated value (if any) in the infomaps. Due to the interdependence of the ti and in commands (a .ti +n is a temporary indent with respect to the current indent value), these are dealt with in a single pass. The order in which each setting pass is done is irrelevent.

During these passes, those commands which are specified to implicitly cause a break cause the insertion of a br command.

Most of the commands have served their purpose once they have set some value and perhaps inserted br commands, and so can be discarded.

Output from this stage is list porc, where porc is defined as:

DATA porc <= ptext(infomap,list char) ++ command.carg

which differs from torc by the extension of text to ptext by the addition of an infomap to store values. The commands remaining are BP, SP and BR, all the others having been removed in this stage.

Stage 3: - Preliminary processing of text lines

At this point we can begin to carry out some actions upon the text lines. We do all that is independent of the formation of output lines or pages. For our present set of commands, this means underlining, and processing empty text lines and text lines which commence with a blank.

UNDERLINE is the procedure to handle underlining: Lines signalled to be underlined (i.e. lines with ULVAL set to true in their infomaps) have their characters (other than blanks, backspaces

and existing underlines) underlined by expanding them to

character BACKSPACE UNDERLINE

BLANKS is the procedure to deal with empty lines or lines starting with a blank: Lines containing nothing but blanks are replaced by a BR command and an empty line with the FIVAL of its infomap set to false (so that this empty line will be output).

Lines commencing with a blank but which do have non-blanks cause the insertion of a BR command, are modified by the removal of all leading blanks, and have their TIVAL set to their INVAL+number of blanks removed.

<u>Stage</u> <u>4</u>: - Formation of lines for output

This stage produces lines ready for packing into pages. The operations which must be done are centering, filling and right justifying, together with dealing appropriately with those lines which do not fit in the indicated margins.

LINES is the top-level procedure to control this. It goes through the list of text and commands doing the following:

If it is a command:

then if it is a br command, discard it

otherwise simply pass it through.

If it is a text line:

There are three possibilities: either it needs centering, or it needs putting out as it is, or it is the first of possibly several more lines to be put out in the "fill" mode. These cases can be determined by examining the CEVAL and FIVAL of the infomap,

remembering that lines to be centred are not filled, whether in fill mode or not.

PUT is the procedure used to put out text lines, taking into account margins. Provided the line will fit into the margins, it can be set ready for output by inserting as many blanks as the TIVAL of the infomap of that line. If it won't fit, then as much of it as will fit in the margins is put out in the first line, and the remaining characters are handed to DEFAULTPUT to deal with.

DEFAULTPUT merely puts out PAGEWIDTH-wide lines until it has exhausted its input.

LINECENTRE is used on those lines requiring centering. This increments the TIVAL of the infomap by half the extra space, the extra being the RMVAL - (TIVAL + width of line), and gives the output to put. If the line is too wide to fit in the margin, this increment will be zero.

The "fill" mode is more tricky, as the input text lines are being used to provide words, which in turn are put into full output lines, right justified. When a line is encountered to be "filled" the following actions take place: GETLINESTOFILL is called on the input. This returns the input line(s) which are to provide the words, and the remainder of the input for LINES to continue working on. GETWDSTOFILL is used to convert the list of lines to a list of words, and finally PUTPARAGRAPH takes this and produces a list of output lines.

GETLINESTOFILL continues ammassing input text lines until either

end-of-input is reached, or a BR command is encountered, which is the signal to break the filling process. No other commands are expected, since only br, sp and bp commands will have survived to this stage, and the sp and bp commands will have had a br command inserted immediately before them.

GETWDSTOFILL, given a list of text lines, produces a list of words. The words are formed from contiguous sequences of non-blank characters in the text lines, blanks and end-of-line considered as separating the words. The infomap associated with a text line is given to each word formed from that line - but the second and beyond words have the TIVAL of the infomap reset to the INVAL. This is to ensure that the (temporary) TIVAL is not propogated into the following words which might appear in different output lines.

PUTPARAGRAPH takes a list of words (each with an infomap) and produces output lines, filling them with the words, and spreading the words out to right-justify them. Because several words, perhaps with _differing_ infomaps, are likely to be included in a single output line, a decision needs to be made on how the parameters of the line (left margin & right margin) are to be determined. The straightforward, and it turns out only logical, convention is to say that the infomap of the _first_ word determines the characteristics for the entire line. An example will illustrate this point:

Suppose we have the following sequence of input:

```
.fi
.rm 20
.in 0
USS Enterprise has sustained
damage to front shields
.rm 12
but enemy losses are high.
.nf
```

Under my convention, this is turned into:

```
USS  Enterprise  has
sustained damage  to
front   shields  but
enemy losses
are    high.
```

Note that the right margin has changed neatly from 20 to 12 between the lines "front shields but" and "enemy losses". However, if, say, the convention that the latest infomap is used to determine characteristics, we might perhaps get:

```
USS  Enterprise  has
sustained damage  to
front shields but
enemy losses
are    high.
```

due to the change in right margin between the words "shields" and "but", the right margin of the line starting "front" is changed midstream to less than the width of the words already in it, resulting in a right margin of width neither 20 nor 12.

If a single word is too wide to fit between the margins, it is given to PUT to deal with, otherwise it and following words are amassed until no more will fit into the available space, and SPREAD is called on these to put them out as a justified line. The remaining few words left when filling comes to an end which do not fill an entire line are put out with single spaces between them, as is the end of this paragraph.

SPREAD takes a list of words, and padds them out with blanks to make them right justified. Of course, if there is only one word, no spreading can be done, but otherwise it calculates the total number of blanks to be added (which will be space available, i.e. RMVAL - TIVAL of first word's infomap, - sum of widths of words), and gives

this value along with the list of words to SUBSPREAD to partition out the blanks.

SUBSPREAD does the "dealing-out" of the blanks to the gaps between the words. While there are more blanks to be added than gaps, it simply extends each gap by one. For the remaining odd few blanks, ADDEXTRASTOGAPS is used to put them in.

ADDEXTRASTOGAPS is called with a truthvalue indicating whether the extra blanks are to be allocated from left-to-right or from right-to-left among the gaps. By alternating the value of this in successive calls, we get the extra blanks distributed at each side of the page.

Output from this stage is a list of porc, where the ptext's are lines ready to be amassed into pages, interspersed with sp and bp commands.

Stage 5: - Formation of pages

In this stage the text lines, interspersed with bp and sp commands, are bunched into pages, complete with header and footer titles.

The outermost procedure, PAGES, initialises the current page number to zero, and calls SUBPAGES to do the work. The action of SUBPAGES is to form a whole page for output, and, if there is more input to be dealt with, calls itself recursively on that.
In a similar manner to the formation of full lines (PUTPARAGRAPH in the preceeding stage) I adopt the convention that the infomap of the

first line to be put in the page determines the characteristics of the entire page. For pages, these are header title and footer title. PUTHDR and PUTFTR are used to produce the header and footer titles respectively.

If SUBPAGES finds a bp command at the start of its input, then it merely resets the page number, and does not produce an entirely blank page. Similarly an sp command at the start is discarded, so that unneccessary blank lines don't start the page.

FILLPAGE builds up the lines to be put in a page, the remainder of the input for SUBPAGES to continue on, and the next page number. It is given the amount of space remaining for the current page, the input, the current line spacing (it will need to know this in order to deal with sp commands), the current page number, and the expected next page number, and acts in the following manner:

If there is no space left, then exactly enough lines to fill the page have already been found, so return.

If end-of-input or a bp command is encountered, BLANKLINES is called with the remaining space value, and generates exactly that many blank lines to fill out to the end of the current page. A bp command is used to set the next page number. Then return.

If an sp command is encountered, SKIP is called to space down the appropriate number of lines. The space remaining is decremented by the number of blank lines SKIP produces, and FILLPAGE is called recursively.

SKIP is told the size of the remaining space, the number of lines to be skipped, and the current line-spacing. It generates the minimum of (space remaining, number of lines to be skipped * line

spacing) blank lines.

If a text line is encountered it is passed without modification and the minimum of (space remaining after text line, current line spacing - 1) blank lines are generated after it. FILLPAGE is called recursively with the remaining space and input, and the latest text line's LSVAL.

6.3 NPL PROTOPROGRAM

```
DEF
///  ********************  DATA TYPES   ********************
///  ********************     AND       ********************
///  ********************  UTILITIES    ********************

///                 TRUTHVALUES - assume =,/=,not,and,or
                          already defined.

VAR FLAG,LTOR : truval

///                         LISTS - length for lists:

+++ length(list atom) <= num
VAR ATM : atom   VAR ATML : list atom
--- length(NIL) <= 0
--- length(ATM::ATML) <= succ length(ATML)

///                         NUMBERS & operations on them
                          - assume + already defined.

VAR N,M,MAXW,CURW,OLDN,DEFAULT,MINN,MAXN,NEWN,VAAL,VALIN,VALTI,
    COUNT,BLANKCOUNT,TOTALWIDTH,BCOUNT,GCOUNT,PGNUM,NXTPGNUM,
    CURPGNUM,SKIPCOUNT,VALLS : num

inf 9 *
+++ num * num <= num
--- 0 * M <= 0
--- (succ N) * M <= M + (N * M)

inf 8 -
+++ num - num <= num
--- 0 - M <= 0
--- N - 0 <= N
--- (succ N) - (succ M) <= N - M

+++ half(num) <= num
--- half(0) <= 0
--- half(succ 0) <= 0
--- half(succ succ N) <= succ(half(N))

inf 4 >>
+++ num >> num <= truval
--- 0 >> M <= false
--- (succ N) >> 0 <= true
--- (succ N) >> (succ M) <= N >> M

+++ min(num,num) <= num
--- min(N,M) <= N if M >> N
             <= M ifnot

+++ max(num,num) <= num
--- max(N,M) <= M if M >> N
```

```
          <= N ifnot

inf 9 //
+++ num // num <= num
--- N // 0 <= UNDEF
--- N // (succ M) <= 0 if (succ M) >> N
                  <= succ((N - (succ M)) // (succ M)) ifnot

inf 9 rem
+++ num rem num <= num
--- N rem 0 <= UNDEF
--- N rem (succ M) <= N - ((N // (succ M)) * (succ M))


+++ HUGE <= num   --- HUGE <= 25

END

DEF

/// *******************  CHARACTERS etc.  ********************

DATA char <= CH0 ++ CH1 ++ CH2 ++ CH3 ++ CH4 ++ CH5 ++ CH6 ++ CH7 ++
             CH8 ++ CH9 ++ CHA ++ CHB ++ CHC ++ CHD ++ CHE ++ CHF ++
             CHG ++ CHH ++ CHI ++ CHJ ++ CHK ++ CHL ++ CHM ++ CHN ++
             CHO ++ CHP ++ CHQ ++ CHR ++ CHS ++ CHT ++ CHU ++ CHV ++
             CHW ++ CHX ++ CHY ++ CHZ ++
             CHUNDERLINE ++ CHBACKSPACE ++
             CHAPOSTROPHE ++ CHDOT ++ CHPLUS ++ CHMINUS ++ CHBLANK
             CHHASH

VAR C,C1,C2 : char
VAR CL,CL1,REMCL,FIRSTCL,CLSOFAR,TITLE : list char

DATA line <= lin(list char)

VAR BLINES,PAGELINES,MORELINES,LINL : list line

///                   Digits from/to characters

+++ digitval(char) <= num
--- digitval(C) <= 0 if C=CH0
                <= 1 if C=CH1
                <= 2 if C=CH2
                <= 3 if C=CH3
                <= 4 if C=CH4
                <= 5 if C=CH5
                <= 6 if C=CH6
                <= 7 if C=CH7
                <= 8 if C=CH8
                <= 9 if C=CH9
                <= UNDEF ifnot

+++ digit(num) <= char
--- digit(0) <= CH0
```

```
--- digit(1) <= CH1
--- digit(2) <= CH2
--- digit(3) <= CH3
--- digit(4) <= CH4
--- digit(5) <= CH5
--- digit(6) <= CH6
--- digit(7) <= CH7
--- digit(8) <= CH8
--- digit(9) <= CH9
--- digit(succ succ succ succ succ succ succ succ succ succ N) <=
 UNDEF


///          Conversion between character lists and numbers

+++ cltonum(list char) <= num
+++ subcltonum(list char,num) <= num
--- cltonum(nil) <= UNDEF
--- cltonum(C::CL) <= subcltonum(C::CL,0) if digitval(C)/=UNDEF
                   <= UNDEF ifnot

--- subcltonum(nil,N) <= N
--- subcltonum(C::CL,N) <= N if C=CHBLANK
                        <= UNDEF if digitval(C)=UNDEF
                         <= subcltonum(CL,(N*10)+digitval(C)) ifnot

+++ numtocl(num) <= list char
--- numtocl(N) <= [digit(N rem 10)] if N // 10 = 0
               <= numtocl(N//10) <> [digit(N rem 10)] ifnot


///                    Skipping blanks and non-blanks

+++ skipblanks(list char) <= list char
--- skipblanks(nil) <= nil
--- skipblanks(C::CL) <= skipblanks(CL) if C=CHBLANK
                      <= C::CL ifnot

+++ skipalphas(list char) <= list char
--- skipalphas(nil) <= nil
--- skipalphas(C::CL) <= C::CL if C=CHBLANK
                      <= skipalphas(CL) ifnot


///                    Width of character lists - handling backspace

+++ width(list char) <= num
+++ subwidth(list char,num,num) <= num
--- width(CL) <= subwidth(CL,0,0)

--- subwidth(nil,MAXW,CURW) <= MAXW
--- subwidth(C::CL,MAXW,CURW)
                    <= subwidth(CL,MAXW,CURW-1) if C=CHBACKSPACE
                    <= subwidth(CL,max(MAXW,CURW+1),CURW+1) ifnot
```

```
END


DEF

/// *****************  COMMANDS and their arguments  *******

///            carg - possible arguments of commands

DATA carg <= nullarg ++ string(list char) ++
             signed(char,num) ++ unsigned(num)

VAR CAR : carg

+++ stringof(carg) <= list char
--- stringof(string(CL)) <= CL

///            Commands

DATA command <= BR ++ BP ++ CE ++ FI ++ FO ++ HE ++ IN ++
                LS ++ NF ++ PL ++ RM ++ SP ++ TI ++ UL ++
                UNKNOWN

VAR CMD : command


///                             INFOMAPS
                        Values which may occur in them:

DATA ival <= ivc(list char) ++ ivn(num) ++ ivt(truval)

VAR IV : ival

+++ clof(ival) <= list char
--- clof(ivc(CL)) <= CL
+++ numof(ival) <= num
--- numof(ivn(N)) <= N
+++ tof(ival) <= truval
--- tof(ivt(FLAG)) <= FLAG

///              Infomap types of values

DATA itype <= PLVAL ++ INVAL ++ TIVAL ++ CEVAL ++ FIVAL ++
              ULVAL ++ LSVAL ++ HEVAL ++ FOVAL ++ RMVAL

VAR IT,IT1 : itype


///                Infomaps

DATA infomap <= im(list(tuple2(itype,ival)))

VAR IIL : list tuple2(itype,ival)
VAR IMAP,NEWIMAP : infomap
```

```
+++ NULLMAP <= infomap
--- NULLMAP <= im(NIL)

+++ addtomap(infomap,itype,num) <= infomap
+++ addctomap(infomap,itype,list char) <= infomap
+++ addttomap(infomap,itype,truval) <= infomap
--- addtomap(im(IIL),IT,N) <= IM(<IT,ivn(N)> ::IIL)
--- addctomap(im(IIL),IT,CL) <= im(<IT,ivc(CL)> ::IIL)
--- addttomap(im(IIL),IT,FLAG) <= im(<IT,ivt(FLAG)> ::IIL)

inf 10 zz  inf 10 zzc  inf 10 zzt
+++ infomap zz itype <= num
+++ infomap zzc itype <= list char
+++ infomap zzt itype <= truval
--- im(NIL) zz IT <= UNDEF
--- im(NIL) zzc IT <= UNDEF
--- im(NIL) zzt IT <= UNDEF
--- im(<IT1,IV> ::IIL) zz IT <= numof(IV) if IT=IT1
                              <= im(IIL) zz IT ifnot
--- im(<IT1,IV> ::IIL) zzc IT <= clof(IV) if IT=IT1
                               <= im(IIL) zzc IT ifnot
--- im(<IT1,IV> ::IIL) zzt IT <= tof(IV) if IT=IT1
                               <= im(IIL) zzt IT ifnot


///              text or command

inf 7 $
DATA torc <= text(list char) ++ command $ carg

VAR TCL : list torc


///            text with infomap or command

inf 7 .

DATA porc <= ptext(infomap,list char) ++ command.carg

VAR PCL,PCLREM : list porc

///            word with infomap

DATA pword <= wd(infomap,list char)

VAR W,W1 : pword
VAR PWL,REMPWL,WLSOFAR : list pword

END

DEF

/// ****************   PRESETTING CONSTANTS   *****************
```

```
+++ PAGELEN <= num     --- PAGELEN <= 10
+++ PAGEWIDTH <= num  --- PAGEWIDTH <= 2*10
+++ PGNUMCHAR <= char --- PGNUMCHAR <= CHHASH
+++ CHCOMMAND <= char --- CHCOMMAND <= CHDOT
+++ HDR1 <= num  --- HDR1 <= 1
+++ HDR2 <= num  --- HDR2 <= 2
+++ FTR1 <= num  --- FTR1 <= 1
+++ FTR2 <= num  --- FTR2 <= 2
+++ HDRLENGTH <= num  --- HDRLENGTH <= HDR1+HDR2
+++ FTRLENGTH <= num  --- FTRLENGTH <= FTR1+FTR2


END

DEF

/// ******************                          ********************
/// ********************   COMMAND DECODING   ********************
/// ******************                          ********************


+++ gettitle(list char) <= carg
+++ subgettitle(list char) <= list char
--- gettitle(CL) <= string(subgettitle(skipblanks(skipalphas(CL))))

--- subgettitle(nil) <= nil
--- subgettitle(C::CL) <= CL if C=CHAPOSTROPHE
                      <= C::CL ifnot


+++ getnum(list char) <= carg
+++ subgetnum(list char) <= carg
--- getnum(CL) <= subgetnum(skipblanks(skipalphas(CL)))

--- subgetnum(nil) <= NULLARG
--- subgetnum(C::CL)
         <= NULLARG if (C=CHPLUS or C=CHMINUS) and cltonum(CL)=UNDEF
         <= signed(C,cltonum(CL)) if (C=CHPLUS or C=CHMINUS)
         <= NULLARG if cltonum(C::CL)=UNDEF
         <= unsigned(cltonum(C::CL)) ifnot


+++ cdecode(list char) <= tuple2(command,carg)

--- cdecode(nil) <= <UNKNOWN,NULLARG>
--- cdecode(C1::nil) <= <UNKNOWN,NULLARG>
--- cdecode(C1::(C2::CL))
     <= <BR,NULLARG>        if C1=CHB and C2=CHR
     <= <BP,getnum(CL)>     if C1=CHB and C2=CHP
     <= <CE,getnum(CL)>     if C1=CHC and C2=CHE
     <= <FI,NULLARG>        if C1=CHF and C2=CHI
     <= <FO,gettitle(CL)>   if C1=CHF and C2=CHO
     <= <HE,gettitle(CL)>   if C1=CHH and C2=CHE
     <= <IN,getnum(CL)>     if C1=CHI and C2=CHN
     <= <LS,getnum(CL)>     if C1=CHL and C2=CHS
```

```
    <= <NF,NULLARG>       if C1=CHN and C2=CHF
    <= <PL,getnum(CL)>    if C1=CHP and C2=CHL
    <= <RM,getnum(CL)>    if C1=CHR and C2=CHM
    <= <SP,getnum(CL)>    if C1=CHS and C2=CHP
    <= <TI,getnum(CL)>    if C1=CHT and C2=CHI
    <= <UL,getnum(CL)>    if C1=CHU and C2=CHL
    <= <UNKNOWN,NULLARG> ifnot


+++ decode(list line) <= list torc

--- decode(nil) <= nil
--- decode(lin(nil)::LINL) <= text(nil)::decode(LINL)
--- decode(lin(C::CL)::LINL) <=
              cond(C=CHCOMMAND,
                   cond(CMD=UNKNOWN,
                        decode(LINL),
                        (CMD$CAR)::decode(LINL))
                     where <CMD,CAR> == cdecode(CL),
                   text(C::CL)::decode(LINL))
END


DEF

/// ********************* INITIALISING INFOMAPS ************

+++ initmap(list torc) <= list porc
--- initmap(nil) <= nil
--- initmap(text(CL)::TCL) <= ptext(NULLMAP,CL)::initmap(TCL)
--- initmap(CMD$CAR::TCL)  <= CMD.CAR::initmap(TCL)

END

DEF
/// ******************                              *****************
/// ****************** DEALING WITH COMMANDS         *****************
/// ******************                              *****************

///            Set - to set numeric values from existing value,
               argument of command, default and limits.

+++ sset(num,carg,num,num,num) <= num

--- sset(OLDN,NULLARG,DEFAULT,MINN,MAXN) <= DEFAULT
--- sset(OLDN,unsigned(NEWN),DEFAULT,MINN,MAXN) <=
                                  min(MAXN,max(MINN,NEWN))
--- sset(OLDN,signed(CHPLUS,NEWN),DEFAULT,MINN,MAXN) <=
                             min(MAXN,max(MINN,OLDN+NEWN))
--- sset(OLDN,signed(CHMINUS,NEWN),DEFAULT,MINN,MAXN) <=
                             min(MAXN,max(MINN,OLDN-NEWN))


///                     PAGE LENGTH
```

```
+++ dopl(list porc) <= list porc
+++ subdopl(num,list porc) <= list porc
+++ INITPL <= num    --- INITPL <= PAGELEN

--- dopl(PCL) <= subdopl(INITPL,PCL)

--- subdopl(VAAL,nil) <= nil
--- subdopl(VAAL,ptext(IMAP,CL)::PCL) <=
                ptext(addtomap(IMAP,PLVAL,VAAL),CL)::subdopl(VAAL,PCL)
--- subdopl(VAAL,CMD.CAR::PCL)
             <= subdopl(sset(VAAL,CAR,PAGELEN,
                            1+HDRLENGTH+FTRLENGTH,PAGELEN),PCL)
                                    if CMD=PL
      <= CMD.CAR::subdopl(VAAL,PCL) ifnot


///                SPACE DOWN and BEGIN PAGE

+++ dosp(list porc) <= list porc
--- dosp(nil) <= nil
--- dosp(ptext(IMAP,CL)::PCL) <= ptext(IMAP,CL)::dosp(PCL)
--- dosp(CMD.CAR::PCL)
      <= BR.NULLARG::(SP.unsigned(sset(0,CAR,1,0,HUGE))::dosp(PCL))
                               if CMD=SP
      <= CMD.CAR::dosp(PCL) ifnot


+++ dobp(list porc) <= list porc
--- dobp(nil) <= nil
--- dobp(ptext(IMAP,CL)::PCL) <= ptext(IMAP,CL)::dobp(PCL)
--- dobp(CMD.CAR::PCL) <= BR.NULLARG::(CMD.CAR::dobp(PCL)) if CMD=BP
                       <= CMD.CAR::dobp(PCL) ifnot


///                    INDENT and TEMPORARY INDENT

+++ doinandti(list porc) <= list porc
+++ subdoinandti(num,num,list porc) <= list porc
+++ INITIN <= num    --- INITIN <= 0
+++ INITTI <= num    --- INITTI <= 0

--- doinandti(PCL) <= subdoinandti(INITIN,INITTI,PCL)

--- subdoinandti(VALIN,VALTI,nil) <= nil
--- subdoinandti(VALIN,VALTI,ptext(IMAP,CL)::PCL) <=
        ptext(addtomap(addtomap(IMAP,INVAL,VALIN),TIVAL,VALTI),CL)
        ::subdoinandti(VALIN,VALIN,PCL)
--- subdoinandti(VALIN,VALTI,CMD.CAR::PCL)
         <= subdoinandti(VAAL,VAAL,PCL)
        where <VAAL> == <sset(VALIN,CAR,0,0,PAGEWIDTH)> if CMD=IN
         <= BR.NULLARG::subdoinandti(VALIN,
                  sset(VALIN,CAR,0,0,PAGEWIDTH),PCL) if CMD=TI
         <= CMD.CAR::subdoinandti(VALIN,VALTI,PCL) ifnot
```

```
///                         CENTRE

+++ doce(list porc) <= list porc
+++ subdoce(num,list porc) <= list porc
+++ INITCE <= num   --- INITCE <= 0

--- doce(PCL) <= subdoce(INITCE,PCL)

--- subdoce(COUNT,nil) <= nil
--- subdoce(0,ptext(IMAP,CL)::PCL) <=
        ptext(addttomap(IMAP,CEVAL,FALSE),CL)::subdoce(0,PCL)
--- subdoce(SUCC COUNT,ptext(IMAP,CL)::PCL) <=
        ptext(addttomap(IMAP,CEVAL,TRUE),CL)::subdoce(COUNT,PCL)
--- subdoce(COUNT,CMD.CAR::PCL)
      <= BR.NULLARG::subdoce(sset(COUNT,CAR,1,0,HUGE),PCL) if CMD=CE
      <= CMD.CAR::subdoce(COUNT,PCL) ifnot


///                        UNDERLINE

+++ doul(list porc) <= list porc
+++ subdoul(num,list porc) <= list porc
+++ INITUL <= num   --- INITUL <= 0

--- doul(PCL) <= subdoul(INITUL,PCL)

--- subdoul(COUNT,nil) <= nil
--- subdoul(0,ptext(IMAP,CL)::PCL) <=
        ptext(addttomap(IMAP,ULVAL,FALSE),CL)::subdoul(0,PCL)
--- subdoul(SUCC COUNT,ptext(IMAP,CL)::PCL) <=
        ptext(addttomap(IMAP,ULVAL,TRUE),CL)::subdoul(COUNT,PCL)
--- subdoul(COUNT,CMD.CAR::PCL)
      <= subdoul(sset(COUNT,CAR,1,0,HUGE),PCL) if CMD=UL
      <= CMD.CAR::subdoul(COUNT,PCL) ifnot


///                 LINE SPACING

+++ dols(list porc) <= list porc
+++ subdols(num,list porc) <= list porc
+++ INITLS <= num   --- INITLS <= 1

--- dols(PCL) <= subdols(INITLS,PCL)

--- subdols(VAAL,nil) <= nil
--- subdols(VAAL,ptext(IMAP,CL)::PCL) <=
        ptext(addtomap(IMAP,LSVAL,VAAL),CL)::subdols(VAAL,PCL)
--- subdols(VAAL,CMD.CAR::PCL)
      <= subdols(sset(VAAL,CAR,1,1,HUGE),PCL) if CMD=LS
      <= CMD.CAR::subdols(VAAL,PCL) ifnot


///                    FILL and NO FILL
```

```
+++ dofi(list porc) <= list porc
+++ subdofi(truval,list porc) <= list porc
+++ INITFI <= truval    --- INITFI <= TRUE

--- dofi(PCL) <= subdofi(INITFI,PCL)

--- subdofi(FLAG,nil) <= nil
--- subdofi(FLAG,ptext(IMAP,CL)::PCL) <=
       ptext(addttomap(IMAP,FIVAL,FLAG),CL)::subdofi(FLAG,PCL)
--- subdofi(FLAG,CMD.CAR::PCL)
       <= BR.NULLARG::subdofi(TRUE,PCL) if CMD=FI
       <= BR.NULLARG::subdofi(FALSE,PCL) if CMD=NF
       <= CMD.CAR::subdofi(FLAG,PCL) ifnot


///                     HEADER

+++ dohe(list porc) <= list porc
+++ subdohe(list char,list porc) <= list porc
+++ INITHE <= list char    --- INITHE <= nil

--- dohe(PCL) <= subdohe(INITHE,PCL)

--- subdohe(TITLE,nil) <= nil
--- subdohe(TITLE,ptext(IMAP,CL)::PCL) <=
        ptext(addctomap(IMAP,HEVAL,TITLE),CL)::subdohe(TITLE,PCL)
--- subdohe(TITLE,CMD.CAR::PCL)
        <= subdohe(stringof(CAR),PCL) if CMD=HE
        <= CMD.CAR::subdohe(TITLE,PCL) ifnot


///                     FOOTER

+++ dofo(list porc) <= list porc
+++ subdofo(list char,list porc) <= list porc
+++ INITFO <= list char    --- INITFO <= nil

--- dofo(PCL) <= subdofo(INITFO,PCL)

--- subdofo(TITLE,nil) <= nil
--- subdofo(TITLE,ptext(IMAP,CL)::PCL) <=
        ptext(addctomap(IMAP,FOVAL,TITLE),CL)::subdofo(TITLE,PCL)
--- subdofo(TITLE,CMD.CAR::PCL)
        <= subdofo(stringof(CAR),PCL) if CMD=FO
        <= CMD.CAR::subdofo(TITLE,PCL) ifnot


///                     RIGHT MARGIN

+++ dorm(list porc) <= list porc
+++ subdorm(num,list porc) <= list porc
+++ INITRM <= num    --- INITRM <= PAGEWIDTH

--- dorm(PCL) <= subdorm(INITRM,PCL)
```

```
--- subdorm(VAAL,nil) <= nil
--- subdorm(VAAL,ptext(IMAP,CL)::PCL) <=
          ptext(addtomap(IMAP,RMVAL,VAAL),CL)::subdorm(VAAL,PCL)
--- subdorm(VAAL,CMD.CAR::PCL)
     <= subdorm(sset(VAAL,CAR,PAGEWIDTH,1,PAGEWIDTH),PCL) if CMD=RM
     <= CMD.CAR::subdorm(VAAL,PCL) ifnot


///              DOCOMMANDS

+++ docommands(list porc) <= list porc

--- docommands(PCL) <=
        dopl(dosp(dobp(doinandti(doce(
        doul(dols(dofi(dohe(dofo(dorm(PCL)))))) )))))


END


DEF
/// ******************                    ******************
/// ******************  INTERMEDIATE  ******************
/// ******************                    ******************

///          'Processing of text lines which are null or start
             with blanks, and performing underlining
             where necessary.@

/// ****************  BLANKS  ****************

+++ blanks(list porc) <= list porc
+++ subblanks(porc) <= porc
+++ delleadblanks(list char) <= tuple2(num,list char)

--- blanks(nil) <= nil
--- blanks(CMD.CAR::PCL) <= CMD.CAR::blanks(PCL)
--- blanks(ptext(IMAP,nil)::PCL) <= BR.NULLARG::
                     (subblanks(ptext(IMAP,nil))::blanks(PCL))
--- blanks(ptext(IMAP,C::CL)::PCL)
         <= BR.NULLARG::(subblanks(ptext(IMAP,C::CL))::
                             blanks(PCL)) if C=CHBLANK
         <= ptext(IMAP,C::CL)::blanks(PCL) ifnot

--- subblanks(ptext(IMAP,CL))
         <= ptext(addttomap(IMAP,FIVAL,FALSE),nil)
           if REMCL=nil
         where <BLANKCOUNT,REMCL> == delleadblanks(CL)
         <= ptext(addtomap(IMAP,TIVAL,MIN(PAGEWIDTH,
                (IMAP zz INVAL)+BLANKCOUNT)),REMCL)
           where <BLANKCOUNT,REMCL> == delleadblanks(CL) ifnot
```

```
--- delleadblanks(nil) <= <0,nil>
--- delleadblanks(C::CL)
       <= <SUCC BLANKCOUNT,REMCL> where <BLANKCOUNT,REMCL> ==
            delleadblanks(CL) if C=CHBLANK
       <= <0,C::CL> ifnot


/// ***************** UNDERLINE *****************

+++ underline(list porc) <= list porc
+++ ulchars(list char) <= list char

--- underline(nil) <= nil
--- underline(CMD.CAR::PCL) <= CMD.CAR::underline(PCL)
--- underline(ptext(IMAP,CL)::PCL)
       <= ptext(IMAP,ulchars(CL))::underline(PCL) if IMAP zzt ULVAL
       <= ptext(IMAP,CL)::underline(PCL) ifnot


--- ulchars(nil) <= nil
--- ulchars(C::CL)
       <= C::ulchars(CL) if C=CHBLANK or C=CHBACKSPACE
                              or C=CHUNDERLINE
       <= C::(CHBACKSPACE::(CHUNDERLINE::ulchars(CL))) ifnot


/// ******************** INTERMEDIATE ******************

+++ intermediate(list porc) <= list porc

--- intermediate(PCL) <= underline(blanks(PCL))

END


DEF

/// ********************        *************************
/// ******************** LINES  *************************
/// ********************        *************************


DATA gorw <= gwd(pword) ++ gap(num)

VAR GWL : list gorw

///            Minor functions:

+++ wltocl(list pword) <= list char
/// wltocl converts list of words to list of characters, inserting
    a blank between each pair of words.
--- wltocl(nil) <= nil
--- wltocl(wd(IMAP,CL)::nil) <= CL
--- wltocl(wd(IMAP,CL)::(W::PWL)) <= CL<>(CHBLANK::wltocl(W::PWL))
```

```
+++ mkblanks(num) <= list char
/// mkblanks(N) <= list of N blanks
--- mkblanks(0) <= nil
--- mkblanks(succ N) <= CHBLANK::mkblanks(N)


+++ gapcount(list gorw) <= num
/// gapcount(GWL) <= number of gaps in GWL
--- gapcount(nil) <= 0
--- gapcount(gwd(W)::GWL) <= gapcount(GWL)
--- gapcount(gap(N)::GWL) <= succ gapcount(GWL)


+++ sumofwidths(list pword) <= num
/// sumofwidths(PWL) <= sum of widths of words of PWL
--- sumofwidths(nil) <= 0
--- sumofwidths(wd(IMAP,CL)::PWL) <= width(CL)+sumofwidths(PWL)


+++ convtocl(list gorw) <= list char
/// convtocl(GWL) converts gaps and words to list of characters
--- convtocl(nil) <= nil
--- convtocl(gwd(wd(IMAP,CL))::GWL) <= CL<>convtocl(GWL)
--- convtocl(gap(N)::GWL) <= mkblanks(N)<>convtocl(GWL)



+++ initgaps(list pword) <= list gorw
/// initgaps(PWL) puts empty gaps between words
--- initgaps(nil) <= nil
--- initgaps(W::nil) <= [gwd(W)]
--- initgaps(W::(W1::PWL)) <= gwd(W)::(gap(0)::initgaps(W1::PWL))


+++ addtogaps(list gorw) <= list gorw
/// addtogaps(GWL) increases length of each gap by one
--- addtogaps(nil) <= nil
--- addtogaps(gwd(W)::GWL) <= gwd(W)::addtogaps(GWL)
--- addtogaps(gap(N)::GWL)  <= gap(succ N)::addtogaps(GWL)


+++ addextrastogaps(truval,num,list gorw) <= list gorw
+++ subaddextras(num,list gorw) <= list gorw
/// addextrastogaps(LTOR,N,GWL) adds one to length of first N
    gaps (if less than N gaps, then to each gap) from left to right
    if LTOR true,  otherwise from right to left

--- addextrastogaps(TRUE,N,GWL) <= subaddextras(N,GWL)
--- addextrastogaps(FALSE,N,GWL) <= rev(subaddextras(N,rev(GWL)))

--- subaddextras(0,GWL) <= GWL
--- subaddextras(succ N,nil) <= nil
--- subaddextras(succ N,gwd(W)::GWL) <= gwd(W)::subaddextras(succ
 N,GWL)
--- subaddextras(succ N,gap(M)::GWL) <= gap(succ
```

```
 M)::subaddextras(N,GWL)


/// ******************** SPREAD ********************

+++ spread(truval,num,list pword) <= list char
+++ subspread(truval,num,list gorw) <= list gorw
/// spread(LTOR,TOTALWIDTH,PWL) converts list of words to list of
    characters.   TOTALWIDTH is assumed to be at least as large
    as the sum of widths of the  words, plus the number of words -1.
/// subspread(LTOR,BCOUNT,GWL) -- GWL is list of words with gaps
    between them.   BCOUNT is number of blanks to be inserted.
    If there are more gaps than  blanks, call addextras to put these
    extra ones in, otherwise use ADDGAPS to increase all gaps by one,
    decrement BCOUNT by the number of gaps, and call again.

--- spread(LTOR,TOTALWIDTH,PWL)
      <= convtocl(subspread(LTOR,TOTALWIDTH-sumofwidths(PWL),
                 initgaps(PWL))) if length(PWL) >> 1
      <= wltocl(PWL) ifnot

--- subspread(LTOR,BCOUNT,GWL)
      <= addextrastogaps(LTOR,BCOUNT,GWL)  if gapcount(GWL) >> BCOUNT
      <= subspread(LTOR,BCOUNT-gapcount(GWL),addtogaps(GWL)) ifnot


/// ******************** PUT, DEFAULTPUT. ******************

///        Minor functions:

+++ splitncl(num,list char) <= tuple2(list char,list char)
+++ subsplit(num,list char,list char) <= tuple2(list char,list char)
/// splitncl(N,CL) <= <first N characters of CL, remaining ones>

--- splitncl(N,CL) <= subsplit(N,nil,CL)

--- subsplit(N,CLSOFAR,nil) <= <CLSOFAR,nil>
--- subsplit(N,CLSOFAR,C::CL)
        <= <CLSOFAR,C::CL> if width(CLSOFAR<>[C]) >> N
        <= subsplit(N,CLSOFAR<>[C],CL) ifnot


+++ defaultput(infomap,list char) <= list porc
/// defaultput(IMAP,CL) used to put out PAGEwidth-wide lines

--- defaultput(IMAP,nil)  <= nil
--- defaultput(IMAP,C::CL)
    <= ptext(IMAP,FIRSTCL)::defaultput(IMAP,REMCL)
         where <FIRSTCL,REMCL> == splitncl(PAGEwidth,C::CL)
                          if width(C::CL) >> PAGEwidth
   <= [ptext(IMAP,C::CL)]  ifnot
```

```
+++ put(infomap,list char) <= list porc
/// put(IMAP,CL) - if CL will not fit into available space,
    i.e. width(CL) + left margin is greater than right margin,
    then put out as many characters as possible in a normal line,
    and use defaultput on the remainder.

--- put(IMAP,CL)
         <=  ptext(IMAP,mkblanks(IMAP zz TIVAL)<>FIRSTCL)
             ::defaultput(IMAP,REMCL) where <FIRSTCL,REMCL> ==
               splitncl((IMAP zz RMVAL) - (IMAP zz TIVAL),CL)
               if width(CL)+(IMAP zz TIVAL) >> (IMAP zz RMVAL)
         <= [ptext(IMAP,mkblanks(IMAP zz TIVAL)<>CL)] ifnot


/// ********************  GETWDSTOFILL  *****************

DATA ptxt <= ptx(infomap,list char)

VAR PTL,LINESTOFILL : list ptxt

+++ getwdstofill(list ptxt) <= list pword
+++ subgetwds(infomap,list char) <= list pword
+++ sublgetwds(infomap,list char) <= list pword
+++ getwd(list char) <= list char
/// getwdstofill(PCL) <= <list of words to be put into full lines>
/// subgetwds(IMAP,CL) <= list of words formed by characters in CL.
    Words are sequences of non-blank(s). First word has IMAP tied
    to it, remaining words have IMAP modified by resetting TIVAL
    by INVAL tied to them

--- getwdstofill(nil) <= nil
--- getwdstofill(ptx(IMAP,CL)::PTL) <=
              subgetwds(IMAP,skipblanks(CL))<>getwdstofill(PTL)

--- subgetwds(IMAP,nil) <= nil
--- subgetwds(IMAP,C::CL)
         <= wd(IMAP,getwd(C::CL))
            ::sublgetwds(addtomap(IMAP,TIVAL,IMAP zz INVAL),
                    skipblanks(skipalphas(C::CL)))


--- sublgetwds(IMAP,nil) <= nil
--- sublgetwds(IMAP,C::CL)
         <= wd(IMAP,getwd(C::CL))
            ::sublgetwds(IMAP,skipblanks(skipalphas(C::CL)))


--- getwd(nil) <= nil
--- getwd(C::CL) <= nil if C=CHBLANK
              <= C::getwd(CL) ifnot


/// ********************  GETLINESTOFILL  *********************
```

```
+++ getlinestofill(list porc) <= tuple2(list ptxt,list porc)

--- getlinestofill(nil) <= <nil,nil>
--- getlinestofill(CMD.CAR::PCL)
        <= <nil,PCL> if CMD=BR
        <= getlinestofill(PCL) ifnot
--- getlinestofill(ptext(IMAP,CL)::PCL) <= <ptx(IMAP,CL)::PTL,PCLREM>
          where <PTL,PCLREM> == getlinestofill(PCL)


/// ********************** PUTPARAGRAPH *********************

+++ fullline(truval,num,list pword) <= tuple2(list char,list pword)
+++ subfullline(truval,num,list pword,list pword) <=
                                          tuple2(list char,list pword)
/// 'fullline(LTOR,TOTALWIDTH,PWL) <= <list of characters to fill
    TOTALWIDTH, formed by taking as many words as possible from PWL
    which will fit in and padding them out using spread if there are
    some more,    remaining words of PWL>   @

--- fullline(LTOR,TOTALWIDTH,PWL) <=
                          subfullline(LTOR,TOTALWIDTH,nil,PWL)

--- subfullline(LTOR,TOTALWIDTH,WLSOFAR,nil) <= <wltocl(WLSOFAR),nil>
--- subfullline(LTOR,TOTALWIDTH,WLSOFAR,W::PWL)
        <= <spread(LTOR,TOTALWIDTH,WLSOFAR),W::PWL>
                 if width(wltocl(WLSOFAR<>[W])) >> TOTALWIDTH
        <= subfullline(LTOR,TOTALWIDTH,WLSOFAR<>[W],PWL) ifnot


+++ putparagraph(list pword) <= list porc
+++ subputparagraph(truval,list pword) <= list porc
/// putparagraph(PWL) <= paragraph, i.e. list of lines formed by
    filling lines with words of PWL. If a word is wider than space
    allowed for it (right margin - left margin) then use put on that
    word

--- putparagraph(PWL) <= subputparagraph(TRUE,PWL)

--- subputparagraph(LTOR,nil) <= nil
--- subputparagraph(LTOR,wd(IMAP,CL)::PWL)
      <= put(IMAP,CL)<>subputparagraph(not(LTOR),PWL)
         if width(CL)+(IMAP zz TIVAL) >> (IMAP zz RMVAL)
      <= ptext(IMAP,mkblanks(IMAP zz TIVAL)<>CL1)
         ::subputparagraph(not(LTOR),REMPWL)
         where <CL1,REMPWL> ==
              fullline(LTOR,(IMAP zz RMVAL)-(IMAP zz TIVAL),
                                        wd(IMAP,CL)::PWL)  ifnot


/// *********************** LINECENTRE *********************

+++ linecentre(infomap,list char) <= list porc
/// linecentre(IMAP,CL) increments TIVAL of infomap by half the
    extra space, the extra space being
```

```
    (right margin - (left margin + width of line))
    and gives it to put. If the line is too wide for the margins,
    the increment will be zero.
--- linecentre(IMAP,CL) <= put(
        addtomap(IMAP,TIVAL,(IMAP zz TIVAL) +
             half((IMAP zz RMVAL)-((IMAP zz TIVAL)+width(CL)))),CL)


///  **************************  LINES  ******************

+++ lines(list porc) <= list porc
/// 'lines(PCL) for each text line, if centre indicated,
    use linecentre,    otherwise if fill not indicated, use put,
    otherwise use putparagraph on words got by
    getwdstofill(LINESTOFILL).
     Carry on using lines on remainder of PCL.  @

--- lines(nil) <= nil
--- lines(CMD.CAR::PCL) <= lines(PCL) if CMD=BR
                        <= CMD.CAR::lines(PCL) ifnot

--- lines(ptext(IMAP,CL)::PCL)
        <= linecentre(IMAP,CL)<>lines(PCL) if IMAP zzt CEVAL

        <= put(IMAP,CL)<>lines(PCL) if not(IMAP zzt FIVAL)

        <= putparagraph(getwdstofill(LINESTOFILL))<>lines(PCLREM)
             where <LINESTOFILL,PCLREM> ==
                          getlinestofill(ptext(IMAP,CL)::PCL)
                                   ifnot

END


DEF


///  ***********************         **************************
///  ***********************  PAGES  **************************
///  ***********************         **************************

///                    Minor functions:

+++ blanklines(num) <= list line
/// blanklines(N) <= creates N blank lines
--- blanklines(0) <= nil
--- blanklines(succ N) <= lin(nil) :: blanklines(N)


+++ skip(num,carg,num) <= list line
/// skip(N,unsigned(M),VALLS) produces min(N,M*VALLS) blank lines -
    VALLS is line spacing, N is length remaining of current page,
    M is no of lines to be skipped
--- skip(N,unsigned(SKIPCOUNT),VALLS) <=
                         blanklines(min(N,VALLS*SKIPCOUNT))
```

```
/// **********************  FILLPAGE  ********************

+++ fillpage(num,list porc,num,num,num) <=
                            tuple3(list line,list porc,num)
/// fillpage(N,PCL,VALLS,CURPGNUM,NXTPGNUM) <=
          <N lines,remainder of PCL,next page number>
   When a BP command is encountered, blanklines fill remaining
 space.
   When a SP command is encountered, skip produces appropriate
 number
   of blank lines.
   If a text line, put it out followed by (VALLS-1) blank lines.

--- fillpage(0,PCL,VALLS,CURPGNUM,NXTPGNUM) <= <nil,PCL,NXTPGNUM>

--- fillpage(succ N,nil,VALLS,CURPGNUM,NXTPGNUM)
                             <= <blanklines(succ N),nil,NXTPGNUM>

--- fillpage(succ N,CMD.CAR::PCL,VALLS,CURPGNUM,NXTPGNUM)
      <= <blanklines(succ N),PCL,
            sset(CURPGNUM,CAR,1+CURPGNUM,0,HUGE)> if CMD=BP
      <= <BLINES<>PAGELINES,PCLREM,NEWN>
          where <PAGELINES,PCLREM,NEWN> ==
     fillpage((succ N)-length(BLINES),PCL,VALLS,CURPGNUM,NXTPGNUM)
               where <BLINES> == <skip(succ N,CAR,VALLS)> if CMD=SP
      <= fillpage(succ N,PCL,VALLS,CURPGNUM,NXTPGNUM) ifnot

--- fillpage(succ N,ptext(IMAP,CL)::PCL,VALLS,CURPGNUM,NXTPGNUM) <=
       <lin(CL)::BLINES<>MORELINES,PCLREM,NEWN>
            where <MORELINES,PCLREM,NEWN> ==
      fillpage(N-length(BLINES),PCL,IMAP zz LSVAL,CURPGNUM,NXTPGNUM)
           where <BLINES> == <blanklines(min(N,(IMAP zz LSVAL)-1))>


/// ******************  Footers and Headers  *****************

+++ subst(list char,char,list char) <= list char
/// subst(CL,C1,CL1) substitutes CL1 for C1 in CL
--- subst(nil,C1,CL1) <= nil
--- subst(C::CL,C1,CL1) <= CL1<>subst(CL,C1,CL1) if C=C1
                        <= C::subst(CL,C1,CL1) ifnot


+++ puttitle(list char,num) <= line
/// puttitle(CL,PGNUM) replaces occurrences of PGNUMCHAR in CL by
    character string for PGNUM, and if length of all this is longer
   than PAGEWIDTH, discards extra.
--- puttitle(TITLE,PGNUM) <= lin(FIRSTCL)
          where <FIRSTCL,REMCL> == splitncl(PAGEWIDTH,subst(TITLE,
                                 PGNUMCHAR,NUMTOCL(PGNUM)))



+++ puthdr(list char,num) <= list line
--- puthdr(TITLE,PGNUM)
```

```
                <= blanklines(HDR1) if HDR2=0
                <= blanklines(HDR1)
                   <>(puttitle(TITLE,PGNUM)
                      ::blanklines(HDR2-1)) ifnot


+++ putftr(list char,num) <= list line
--- putftr(TITLE,PGNUM)
                <= blanklines(FTR1) if FTR2=0
                <= blanklines(FTR1)
                   <>(puttitle(TITLE,PGNUM)
                      ::blanklines(FTR2-1)) ifnot


/// ********************  PAGES  ********************

data page <= pag(list line)

VAR PGL : list page

+++ pages(list porc) <= list page
+++ subpages(num,list porc) <= list page
/// 'pages builds up complete pages of output
    subpages(PGNUM,PCL)
    if BP command encountered at top, recompute PGNUM
    if any other command encountered at top, discard it
    A page consists of
         header
         lines of page
         footer
    State at start of first line of page determines page number,
    header and footer titles. @

--- pages(PCL) <= subpages(0,PCL)

--- subpages(PGNUM,nil) <= nil
--- subpages(PGNUM,CMD.CAR::PCL)
         <= subpages(sset(PGNUM,CAR,PGNUM+1,0,HUGE),PCL) if CMD=BP
         <= subpages(PGNUM,PCL) ifnot
--- subpages(PGNUM,ptext(IMAP,CL)::PCL) <= pag(
             puthdr(IMAP zzc HEVAL,PGNUM) <>
             PAGELINES      <>
             putftr(IMAP zzc FOVAL,PGNUM)    ) ::
       subpages(NXTPGNUM,PCLREM)
          where <PAGELINES,PCLREM,NXTPGNUM> ==
       fillpage((IMAP zz PLVAL)-HDRLENGTH-FTRLENGTH,
                 ptext(IMAP,CL)::PCL,(IMAP zz LSVAL),PGNUM,1+PGNUM)

END


DEF
```

```
/// ************************          ************************
/// ************************  OUTPUT  ************************
/// ************************          ************************

+++ output(list page) <= list line

--- output(nil) <= nil
--- output(pag(LINL)::PGL) <= LINL <> output(PGL)

END


DEF

/// ****************                              **************
/// ****************  Top level procedure FORMAT  **************
/// ****************                              **************

+++ format(list line) <= list line

--- format(LINL) <= output(pages(lines(
           intermediate(docommands(initmap(decode(LINL)))))))

END
```

6.4 EVALUATION OF TEXT FORMATTING PROGRAMS

Some questions we may ask of a text formatting program are:

Does the program satisfy the informal specification ?

Where the informal specification is ambiguous, how easy is it to determine from the program how it will behave ?

Can the program be readily modified should we change/extend our text formatting operations ?

The answers to these should reveal just how well designed a program we have. Admittedly, it is a little unfair to expect Kernighan & Plauger's program to be as transparent and flexible as a program written without efficiency considerations in mind, but this is the benefit to be gained from the transformational approach to program development.

6.4.1 Satisfying Informal Specification

I believe that my program <u>does</u> satisfy the informal specification, and furthermore, claim that its overall simplicity of design makes it easier to convince ourselves that this is true.

Kernighan & Plauger's program (which from now on I shall refer to as FORMAT - their name for it), in the main satisfies the informal specification, but not absolutely. Minor differences include:

The peculiarities which may ocur when decreasing right margin size when in "fill" mode (as mentioned in the preceeding section, Stage 4). FORMAT risks the possibility of a line being put out with a margin somewhere between the old value and new, smaller, value.

Temporary indent, caused by a .ti +n command, is specified to

temporarily indent relative to the current indent. Thus

```
.in 5
.ti +3
.ti +4
```

we would expect to cause the next line to be indented by 9 (5+4) spaces, the last ti command signifying the temporary indent. However, FORMAT would indent it by 12 (5+3+4) spaces, adding up both ti commands' relative values.

These may seem trivial differences, and I may be wrong in claiming them as errors, but the main danger lies in their existence being so deeply buried in the complexities of the program.

Spread is a good example of a portion which <u>can</u> be programmed simply, but if written immediately as a single pass, as in FORMAT, is hard to understand. The authors themselves say "This code is tricky (which is not a compliment), but it performs an elaborate function and performs it correctly."

6.4.2 Resolving Abiguities In The Informal Specification

The informal specification is by no means precise, and many ambiguities exist which the programmer must resolve. Where some choice is clearly the expected one from the users point of view, that should be made. If the choice is purely arbitrary, then the easiest or most logical choice from the programmer's point of view should be made. In either case, it should be possible to determine by examining the program exactly how it will act in such circumstances.

Setting of values is an example of how our programs differ: Since FORMAT is processing text and commands as it encounters them,

it takes advantage of this to check the setting of values such as right margin and left margin against each other. Thus the right margin is never set to less than the current left margin, e.g.

If the right and left margins are currently 60 and 40 respectively, then:

```
.rm 20
.in 0
```

would, in FORMAT, cause the right margin to be set to 40, because this was the value of the left margin at the time the ".rm 20" command was received, whereas my program merely sets it to 20 without worry. Clearly the user should not have to worry about the order in which he changes margins.

The advantage of splitting the task into the several stages is that any ambiguity can be resolved by concentrating only on the stage dealing with it. Indeed, each individual stage is constructed in a simple manner, liberally using many small functions to perform easily comprehended actions. FORMAT, although well-designed and structured, nevertheless is noticeably harder to comprehend in its detailed operation. In its design there is no mention of any convention corresponding to the ones I adopt for filling lines and pages. This is probably because the program has been designed only for the mode of operation of going once through the input and dealing with things as soon as they are encountered. Whilst this ensures a degree of efficiency, it does seem to restrict ones thinking.

6.4.3 Changing/Extending The Program

A major test of the flexibility of a program is how easy is it to change or extend the operation of that program.

Since my program is designed around the conceptual stages of text formatting, I claim that provided the changes do not fundamentally alter my underlying concepts, then they will be not too difficult to incorporate. FORMAT, by virtue of its commital to the efficient organisation, is bound to be more restrictive.

Two example changes will illustrate this:

Suppose we wish to modify the way the extra few blanks are partitioned through a filled line, (at present they are distributed alternately from left-to-right and right-to-left on successive lines), perhaps distributing them randomly. In my program, the only change is to procedure ADDEXTRASTOGAPS. In FORMAT this activity is mixed in with that of distributing all the blanks, so the contemplated change effects more than is necessary.

One of Kernighan & Plauger's own suggestions is to enlarge the program to provide multi-column output.

My program should extend to this in a straightforward manner: most of the changes will be confined to the page formation stage, with minor changes to the decoding commands and setting extra values in infomap(s) to handle new commands. Perhaps some of the line formation functions will need modification (e.g. DEFAULTPUT to put out sub-multiples of PAGEWIDTH wide lines when a line for multi-column output is expected from it), but in all these cases, the changes, and the reasons for them, are easy to see.

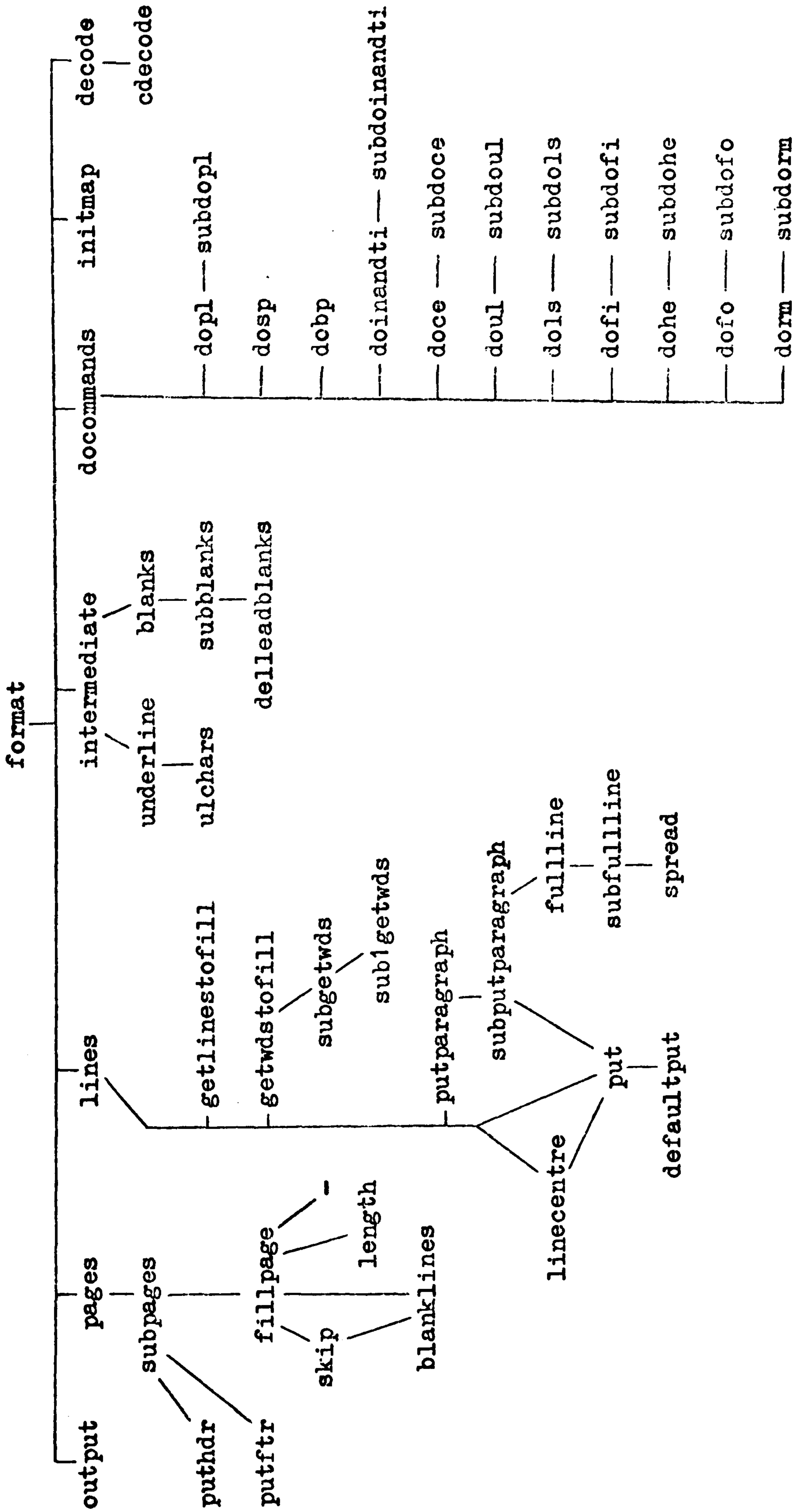
In FORMAT, because the different operations are more intermixed (e.g. the routines to put out page headers and footers get called from within both a routine for putting out a text line and a routine for spacing down lines), the required adjustments will be much harder to determine and carry out correctly.

6.5 TRANSFORMATION TO EFFICIENT VERSION

Transforming the NPL text formatter brings to light new problems due to the large size of the protoprogram. Firstly the equations take up a lot of space if all are to be held in core at once. For the smaller transformations this had never been a serious problem, but here the equations (some 500 of them) consume a prohibitive amount of space. The system had to be adjusted to store the equations in a disc file, and only bring into core such equations as are required for unfolding, when so indicated by the UNFOLD command within a context block. This adjustment fits in well with the concept of context blocks, and makes transformation of the text formatter a practical possibility. The adjustment involved augmenting the system with a package providing functions to write out non-circular structures to disc in character form, and read them back in. Using this, the NPL interpreter deposits equations onto disc as it encounters them, and equation-finding routines now search the disc file rather than an in-core list.

Secondly, the complexity of the whole program is such that it is not possible to see in advance how the transformation will turn out. The gap between protoprogram and anything like Kernighan and Plauger's efficient program is too wide to see across in advance - hence this problem serves as a crucial and unavoidable test of the transformational techniques.

Following the approach to transforming large programs already established, the first step is to construct a diagram of the calling structure of functions within the protoprogram, and from this determine the overall strategy for improvement to be followed.



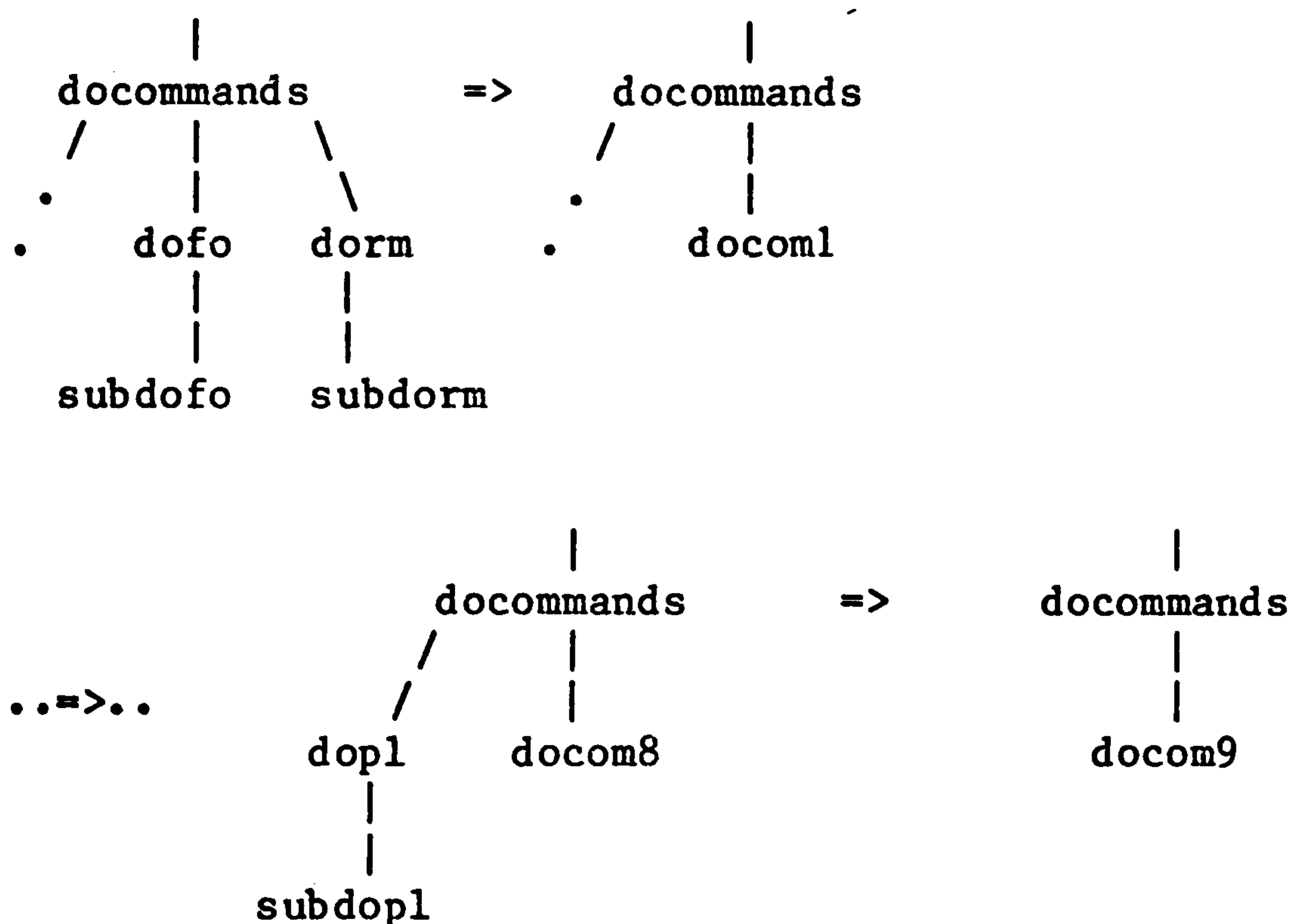
Calling structure of protoprogram

The strategy suggests improving each branch of FORMAT, and then combining improved versions incrementally.

6.5.1 Improving Branches Of FORMAT

The branches which need improvement are DOCOMMANDS, INTERMEDIATE, PAGES and LINES.

6.5.1.1 Improving DOCOMMANDS - Looking at the calling structure of DOCOMMANDS, the strategy suggests incrementally combining functions from inside out (corresponding to left-to-right on the diagram), to arrive at a single function to process all commands.



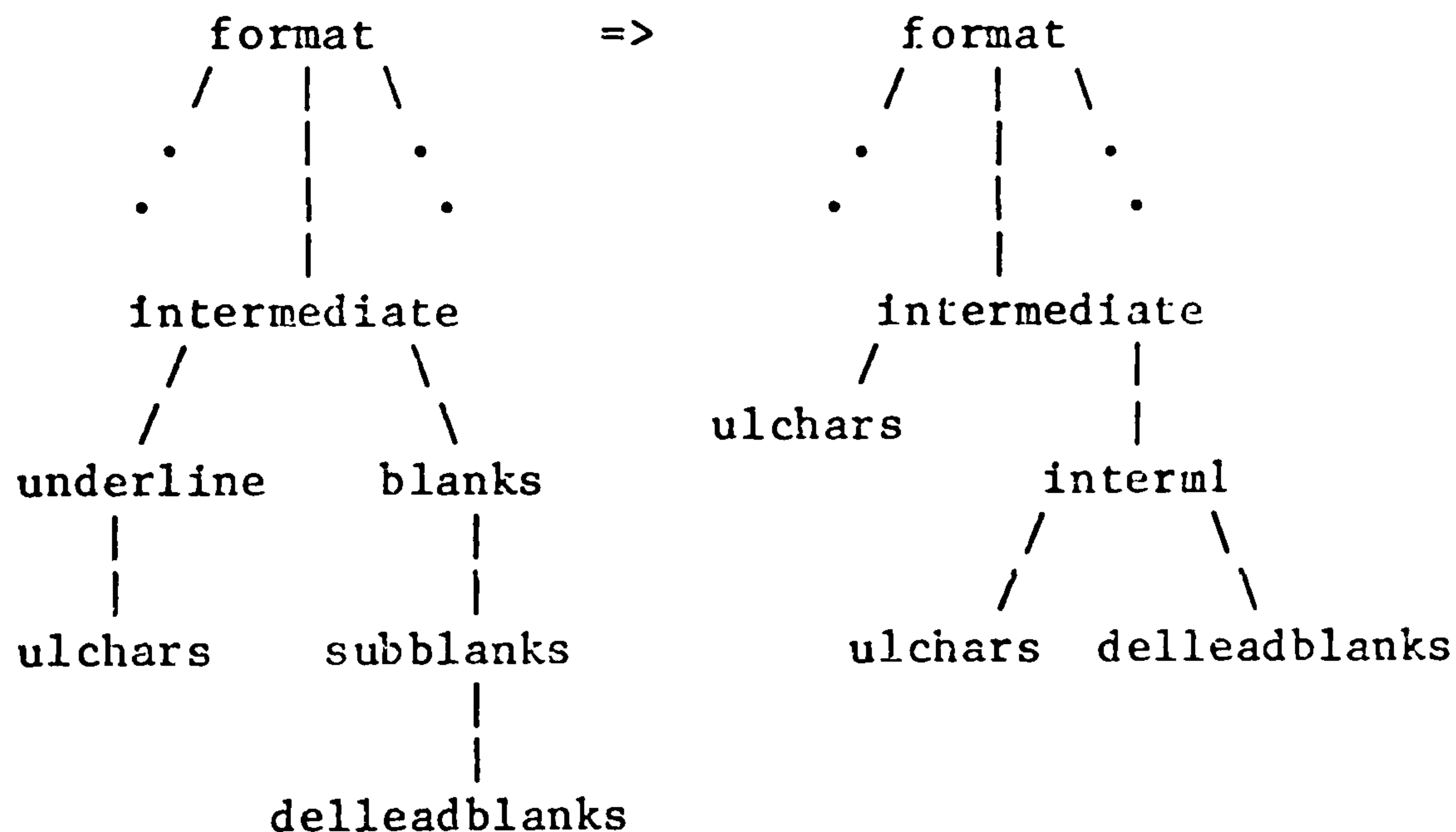
In performing these transformations some disturbing effects become apparent. One is that although the form of the final function is really that of a large (12 branched) cases statement, it has to be modelled in NPL as a deeply nested conditional statement. This is a most unwieldy construct, not easily dealt with by the system. Another difficulty is the abundance of values the final function has

to deal with - these are the values of current right margin, page size and so on. The alternatives are to either maintain these as distinct variables, which consequently have to be passed into each function call (since NPL has no globals), or packaged up into a single information structure with components for each value (analogous to the named common blocks used by Kernighan and Plauger). Although the latter would be the preferable option, it unfortunately clashes with the already unwieldy conditional modelling of the cases statement. The clash is due to the need to make inferences that assigning to one component of the information structure does not upset the value of another component. This obvious property, which would normally fall out automatically by applying the equations for assignment to, and accessing of, the information structures, in this particular context of a deeply nested conditional leads to an explosion in the size of expressions being manipulated. I choose to pass into functions the values in separate variables, tedious in terms of the size of patterns to be specified, but applicable (and in practice once one pattern has been put into a disc file, similar ones can be got by minor edits of that file). The problems exposed here, namely the clumsy modelling of where constructs, and reasoning about assignments to data structures holding several values, are ones which must be tackled and overcome in any future development of the system.

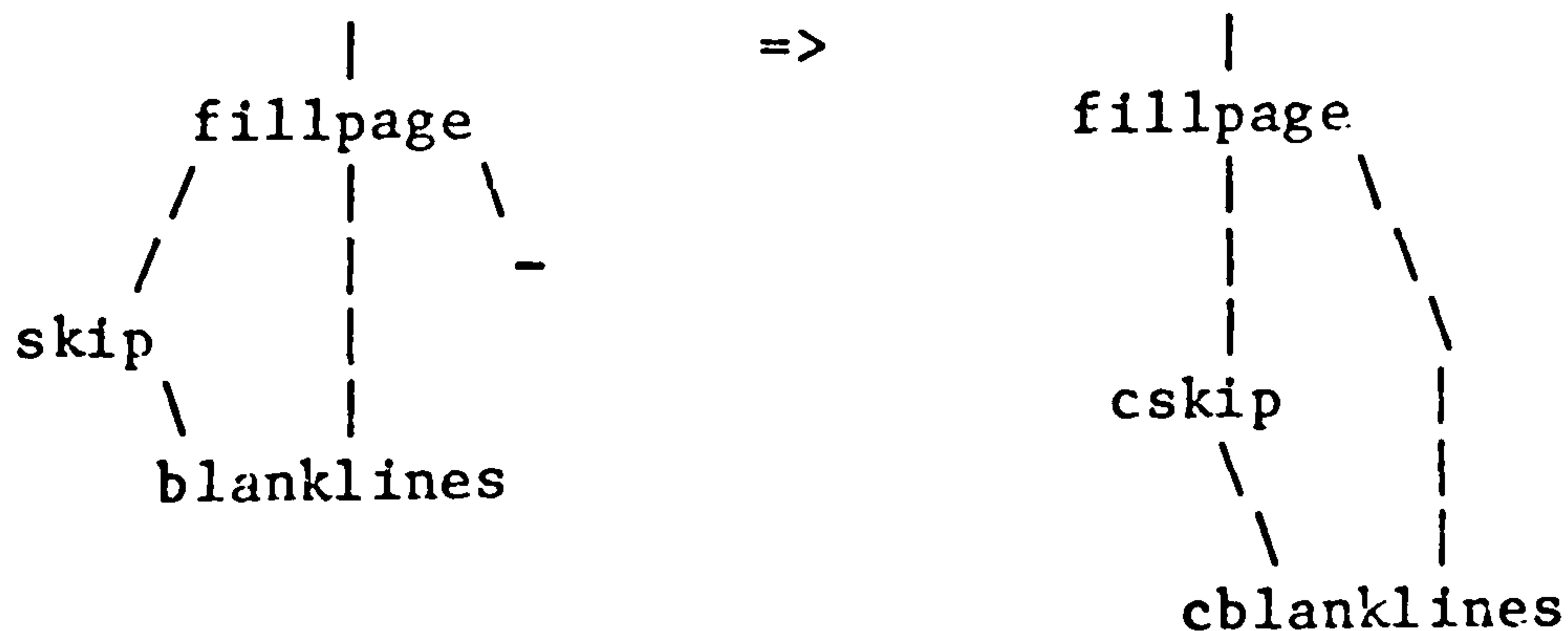
A possible solution to these problems lies in the use of schemata. Currently my patterns provide a form of these suitable for introduction of new functions, and specifying approximately portions of expressions. However in the transformations taking place here there are minor details not crucial to the individual transformations which cannot be expressed within my patterns, and yet are numerous

enough to be troublesome. The use of more conventional schemata should permit these details to be abstracted away. The system could be augmented to transform schemata making use of the existing techniques, i.e. using schemata only to capture the relevant details, without the need to individually verify schematic transformations.

6.5.1.2 Improving INTERMEDIATE - This transformation, a combination of the two branches, goes through without difficulty.



6.5.1.3 Improving PAGES - Here the improvement is to cause skip and blanklines to decrement the count of lines left in the current page as they output blank lines.



6.5.1.4 Improving LINES - The transformation of LINES itself splits into several stages.

Firstly GETWDSTOFILL and GETLINESTOFILL are combined;

```
          |             =>           |
        lines                      lines
       / | \                       / |
      .  |  \                     .  |
     .   |   \                   .   |
         |  getlinestofill           |
         |                           |
     getwdstofill                 lines1
         |                           |
         .                           .
```

Secondly, prior to combining PUTPARAGRAPH with the result of the above step, it is itself improved. At the bottom the function SPREAD requires transformation - however the surrounding structure is such that the internal structure of SPREAD would not be crucial to the transformations to follow. Thus the transformation of SPREAD could be left until later, and done independantly of the rest of the work. What must be considered next is the relationship between SUBPUTPARAGRAPH, FULLLINE and SUBFULLLINE. Currently SUBPUTPARAGRAPH calls FULLLINE as a subroutine to put out a full line of text, and return the remainder of input. In transforming these functions I change the control structure to cause SUBPUTPARAGRAPH to pass control over to a new function, combining FULLLINE and SUBFULLLINE, in the manner of a co-routine rather than a subroutine.

```
        |              =>           |
   putparagraph               putparagraph
        |                           |
        |                           |
 subputparagraph            subputparagraph===paralines
        |                                        |
        |                                        |
    fullline                                   spread
        |                                        |
        |                                        .
  subfullline                                    .
        |
        .
```

This change towards a different calling structure is not suggested by any tactic, and is a clear point in the transformation where I have a choice of what shape of efficient solution to head for. At this point, and at similar points later in the transformation, it is clear that there must be some crucial aspect of the transformation that my patterns, tactics and strategy are failing to capture. The omission seems to lie between the tactics expressing the top level combination of functions, and their realisation in terms of patterns. What neither provide for is the manner in which the calling structures of each half of a combination are merged to form the calling structure of the new function. It is not necessarily the case that this did not arise in transforming the telegram problem or the compiler, rather that they were sufficiently small to be comprehensible all the way through, and any choices made were done unconciously to move towards the efficient solution in mind. Here, however, the inability at this point to see the way through to the final program makes the recognition of such choices unavoidable.

The use of diagrams to show calling structures is more than just a means of illustrating the transformation after the event; in fact these diagrams are of help in observing some of the elusive features of complex combinations, and I make use of them throughout the

transformation process.

The combination of the improved PUTPARAGRAPH and LINES1 follows the coroutine structure established within PUTPARAGRAPH:

```
         |              =>              |
       lines                          lines
      / | \                          / |
     .  |  \                        .  |
    .   |   \                      .   |
        |  lines1                   lines2a===lines2b
        |     \                        |         |
  putparagraph .                       .         .
        |       .
        |
  subputparagraph===paralines
                        |
                        .
```

Functions LINES2A and LINES2B are similar in nature, except one is used when in "fill" mode, the other when not. Hence they could be subsumed by a more general function to carry out the activity of either as indicated by a truthvalue flag to show whether or not in "fill" mode.

```
         |                   =>                   |
       lines                                    lines
      / |                                      / |
     .  |                                     .  |
    .   |                                    .   |
     lines2a===lines2b                        lines2c----
    /   |          |   \                     /   |   \  \
  put  subgetwds   |   spread             put    |    \   spread
                   |                             |     \
              skipblanks                    subgetwds  skipblanks
```

6.5.2 Combining The Improved Branches Of FORMAT

Having completed the improvement of each branch of FORMAT, the strategy suggests combining them in an inside out manner, which in the diagram corresponds to left-to-right:

```
                              format
                                |
 ---------------------------------------------------------------
 |       |       |         |             |             |       |
output  pages  lines  intermediate  docommands  initmap  decode
```

6.5.2.1 Combining INITMAP And DECODE - This small combination is straightforward.

```
      format                 =>        format
     / |   \                          /   |
    .  |    \                        .    |
   .   |     \                      .     |
     initmap  decode                    format1
                |                         |
                |                         |
             cdecode                   cdecode
```

6.5.2.2 Combining FORMAT1 And DOCOMMANDS - This combination is tedious because of the unwieldy nature of DOCOMMANDS, but nevertheless straightforward.

```
      format                 =>        format
     / |   \                          /   |
    .  |    \                        .    |
   .   |     \                      .     |
       |    format1                       |
   docommands  |          format2a====format2====format2b
       |    cdecode                       |
     docom9                            cdecode
```

6.5.2.3 Combining FORMAT2 And INTERMEDIATE - another tedious but straightforward combination. The transformation commands to do this are very similar to those required for the earlier combination of FORMAT1 with DOCOMMANDS, and the combinations during the improvement of DOCOMMANDS itself. The easiest way of creating the file of commands is to edit the file of earlier commands. The patterns I

have developed to guide the system do not capture the relevant details only, which suggests that perhaps the use of some sort of schematic transformations would indeed be a more convenient way of specifying the changes being made.

```
          format
         /  |   \
        .   |    ----------------
       .    |                    \
            |                     \
       intermediate    format2a==format2==format2b
       |    |                     |
       |   interml              cdecode
       | /      \
   ulchars    delleadblanks
```

```
                      format
                     /  |
                    .   |
                   .    |
                        |
          format3a==format3==format3b
                   |        |      \
                cdecode   ulchars   delleadblanks
```

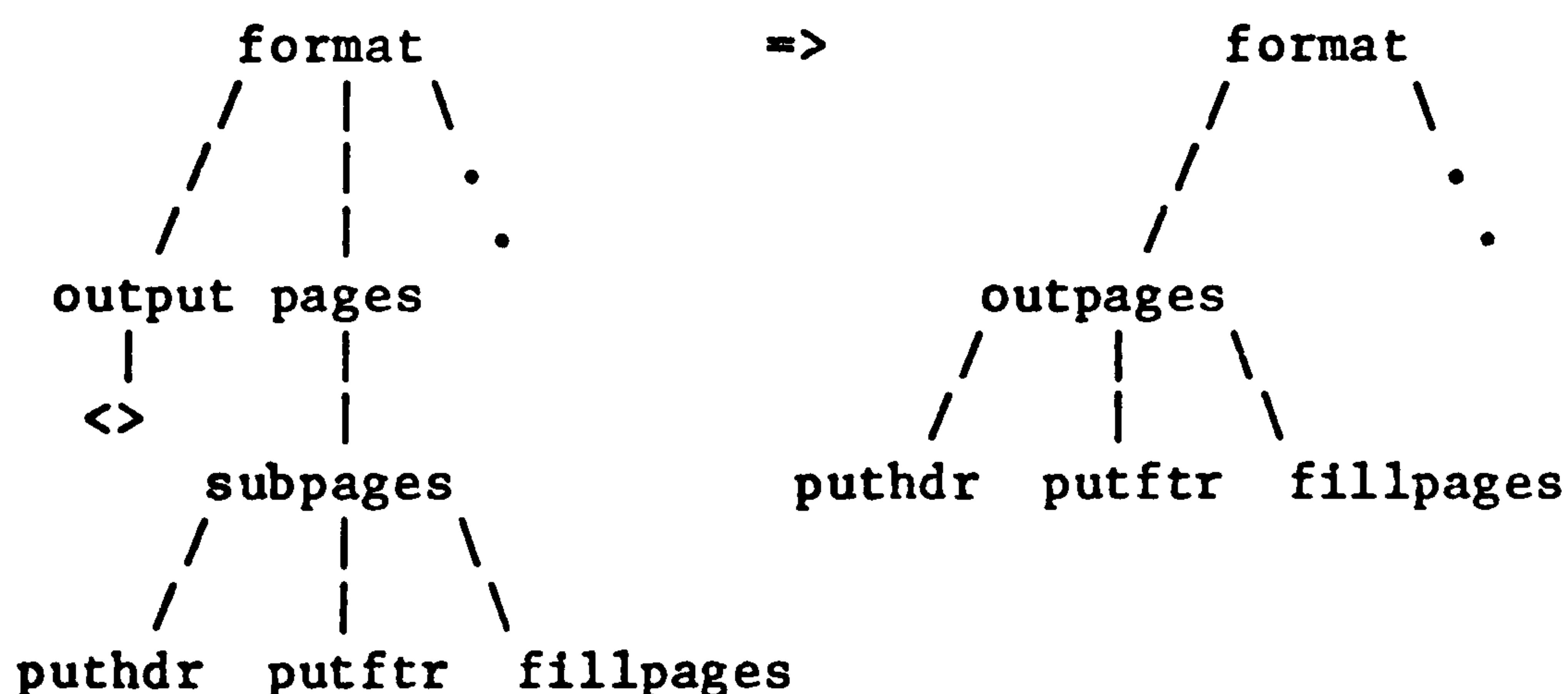
At this point the next step suggested by the strategy would be the combination of FORMAT3 with LINES:

```
             format-----
            /  |  \     \
           /   |   \     \
          /    |    \     \
     output  pages lines format3
               |    |      |
               .    .      .
```

It is apparent, however, that this combination would be hard to perform, because of the unwieldy nature of FORMAT3 and the complex calling structure of LINES. Furthermore, if it were carried out, there would still be the two stages of combining with PAGES and with OUTPUT left to do. These last combinations would be much harder if encumbered by the burden of FORMAT3, so at this point I choose to deviate from the strategy and instead combine LINES with PAGES, then

with OUTPUT, and finally with FORMAT3. Note that the essence of the strategy, that is if one function G uses another function H, improve H before improving G, is not being violated. In this case FORMAT is calling four functions each of which has already been improved, passing the output of one to the input of the next. It is the order in which they are to be combined that I am changing. Viewing the simple strategy as a default to provide a sequence of applications of tactics, we see that in this example the overall transformation follows the strategy most of the time, except here where intuition of the user suggests a better alternative. This illustrates the advantage of defaults over inbuilt techniques - the user can chose to follow a default in the main, overriding it occasionally, wheras an inbuilt technique must either succeed entirely or fail.

6.5.2.4 Combination Of OUTPUT And PAGES - OUTPUT merely strips the stream of pages down to their constituent lines to form the output of the whole process, hence its combination with PAGES is simple.



Here another change in structure seems appropriate. FILLPAGE is being used to return the interior lines of a page for OUTPAGES to surround with the header and footer titles (if any), and the remainder of the input to continue with. Again, a coroutine like

system is introduced so that OUTPAGES starts a new page with the header, and then hands control over to FILLPAGE to carry on from there.

```
         format          =>          format
           |  \                        |  \
           |   .                       |   .
           |    .                      |    .
           |                           |
        outpages                   outpages===filllpage
        /  |   \                       |          |
       /   |    \                      |          |
  puthdr putftr  fillpages         puthdr      putftr
```

6.5.2.5 Combining OUTPAGES And LINES - The combination of these encapsulates much of the complexity of the formatting process. OUTPAGES is concerned with completing pages, whilst LINES builds up the individual lines, handling all the cases of lines to be centred, lines too long to fit in the margins, and fill mode when words are to be squeezed into lines. The result of this combination consists of two main routines, PAGES1 and INPAGES, analogous to OUTPAGES and FILL1PAGE. PAGES1 starts new pages off, and INPAGES carries on once within a page. These use three subroutines, CEINPAGE to centre a line, PUTINPAGE to put out a line and DPUTINPAGE to put out an overlength line.

```
            format       =>       format
           /  |   \                 |   \
          /   |    .                |    .
         /    |     .               |
fillpage==outpages |              pages1===inpages
              |                   |          |  |
            lines                 V          V  |
           /  |  \                |          |  |
          /   |   \               |          |  |
   linecentre |  lines2c          |-ceinpage--|  |
          \   |   /                \    |    /   |
           \  |  /                  putinpage    |
             put                        |    \   |
              |                    dputinpage-->-
            . |
          defaultput
```

6.5.2.6 Combining PAGES1 And FORMAT3 - The program is currently split into two major passes. The first, FORMAT3, does the decoding of incoming commands, setting values in blocks of information to be associated with each input text line, and preliminary processing to handle underlining and blank lines. The second pass, PAGES1, does all the line and page formation processing.

A complete combination of these passes would remove the redundancy associated with creating the intermediate data structure (consisting of lines with associated information, intersperced by commands). However all the difficulties which arose during the earlier transformations would manifest themselves again here, especially those associated with the unwieldy nature of the equations. Hence my last step is to simply model a call-by-need evaluation, defining a new function which would act as FORMAT3 on the input until some intermediate result is obtained, at which point it would behave as PAGES1 applied to this intermediate result. Much, if not all, of the redundancy is removed by this design. The major saving is through not having to process the entire input in a

complete pass and store the entire intermediate structure. Instead the improved version consumes input only as required to produce output, so the storage space will not depend upon the length of the input. The remaining redundancies are small in nature, associated with slight inefficiencies in storing information with each text line and shortly afterwards retrieving it again. Unfortunately the very messy nature of the equations at this point make further improvements, even minor ones, tedious to perform.

```
             format
            /      \
    --------        --------------------
   |                                    |
pages1===inpages            format3a==format3==format3b
|           |  |
V           V  |
|           |  |
|-ceinpage--|  |
 \     |   /   |
  putinpage    |
       |   \   |
 dputinpage-->-
```

```
           format
               |
 ------>----   |           -----<------
|           \  |           /            |
|-pgafmt----pgfmt===inpgfmt----inpgafmt-|
|          /|          |  |\            |
 -pgbfmt--- |          |  | ---inpgbfmt-
            V          V  |
            |          |  |
            |--cefmt---|  |
             \   |    /   |
              putfmt      |
                 |   \    |
              dputfmt--->-
```

A further difficulty which arises during this transformation is that once a function has been defined by cases on one of its arguments, the transformations are unable to generalize that function to a single equation with variable in the argument position.

A trivial example will illustrate this:

```
+++ f(truval) <= truval
--- f(true)   <= true
--- f(false)  <= false
```

Clearly f(T) <= T, but the transformations do not allow us to deduce this. The formatter transformations suffer from this phenomenon, leading to duplication of equations identical except for true/false in a single argument position. Burstall and Darlington mention this limitation in their [1975] paper.

6.6 FINAL PROGRAM

The final program produced as a result of transformation is extremely messy. As shown in the previous diagram, the calling structure is intricate. The functions tend to have a large number of arguments, which serve to further cloud comprehension. Rather than give the entire final program, I present some sample equations, and explain their actions.

```
--- cefmt(false,LLEFT,IMAP,CL,T4,T5,PCL,LINL,T10) <=

     putfmt(false,LLEFT,addtomap(IMAP,TIVAL,IMAP zz TIVAL +
             half(IMAP zz RMVAL) - (IMAP zz TIVAL + width(CL))),
            CL,T4,T5,PCL,LINL,T10)
```

CEFMT is used to centre a line of characters, CL. To do this, it calls PUTFMT with the temporary indent value in the information (IMAP) reset by the appropriate amount. LLEFT is the count of lines remaining on the current page. T4, T5, PCL and T10 hold current values of various parameters, e.g. margins, header and footer titles. LINL is the list of remaining lines of input to continue processing. The first argument of CEFMT and PUTFMT is a truthvalue to indicate whether or not in fill mode - CEFMT is one of the functions that suffers from not being able to generalise distinct true/false cases to a single equation with a variable.

```
--- putfmt(false,0,IMAP,CL,<VALLS,CURPGNUM,NXTPGNUM,VALFO>,
           T5,PCL,LINL,T10) <=

     putftr(VALFO,CURPGNUM) <>
     puthdr(IMAP zzc HEVAL,NXTPGNUM) <>
     putfmt(false,(IMAP zz FLVAL - HDRLENGTH) - FTRLENGTH,IMAP,CL,
            <IMAP zz LSVAL,NXTPGNUM,1+NXTPGNUM,IMAP zzc FOVAL>,
            T5,PCL,LINL,T10)
```

Here PUTFMT has a zero in its second argument position, the count of

lines remaining on the current page. Thus there is no space left, so the footer title is put out, followed by the header of the next page, and PUTFMT is called again at the top of the new page. Note that some of the values are re-set for this new page - the page number becomes the old NXTPGNUM; the line spacing and footer values are got from the IMAP (which is information associated with the line to be put out next); the count of lines remaining becomes the page size less the header and footer lengths.

```
--- pgfmt(false,PGNUM,T5,ptext(IMAP,CL)::PCL,LINL,T10) <=

      cond(IMAP zzt CEVAL,
           puthdr(IMAP zzc HEVAL,PGNUM) <>
           cefmt(false,(IMAP zz PLVAL - HDRLENGTH) - FTRLENGTH,IMAP,
                 CL,<IMAP zz LSVAL,PGNUM,1+PGNUM,IMAP zzc FOVAL>,
                 T5,PCL,LINL,T10),
      cond(not(IMAP zzt FIVAL),
           puthdr(IMAP zzc HEVAL,PGNUM) <>
           putfmt(false,(IMAP zz PLVAL - HDRLENGTH) - FTRLENGTH,
                  <IMAP zz LSVAL,PGNUM,1+PGNUM,IMAP zzc FOVAL>,
                  T5,PCL,LINL,T10),
      pgfmt(true,PGNUM,<subgetwds(IMAP,skipblanks(CL)),true,0,
                        nullmap,[]>,PCL,LINL,T10)
         ))
```

Here PGFMT, the function for starting off a new page, is in no-fill mode (indicated by false as its first argument) and has some intermediate result, ptext(IMAP,CL)::PCL to work on before needing to consider more of the input, LINL. The action it takes is to first see if the line needs centering - if so, the header is put out, and CEFMT will do the rest. If not, and the line is not the start of a paragraph to be filled, then put out the header and call PUTFMT to put out this line. Otherwise, recall PGFMT in fill mode (i.e. with its first argument set to true). The words of the current line are extracted by SUBGETWDS, and together with some other initial values, passed within the 5-tuple so that PGFMT (and other functions) will have the appropriate information needed to fill and justify lines.

When we are at the start of a page, or in the middle of one, and we do not have any intermediate result to work on, PGFMT or INPGFMT must process more of the input.

```
---- pgfmt(FIFLAG,PGNUM,T5,[],lin(C::CL)::LINL,<PLNUM,INNUM,
        TINUM,CENUM,ULNUM,LSNUM,FIFLAG,HETITLE,FOTITLE,RMNUM>) <=

    cond(C=CHCOMMAND,
         pgafmt(FIFLAG,PGNUM,T5,[],U0,U1,LINL,<PLNUM,INNUM,TINUM,
                CENUM,ULNUM,LSNUM,FIFLAG,HETITLE,FOTITLE,RMNUM)
           where <U0,U1> == cdecode(CL) ,
         pgbfmt(FIFLAG,PGNUM,T5,[],nullmap,[],LINL,
                <PLNUM,INNUM,TINUM,CENUM,ULNUM,LSNUM,FIFLAG,
                 HETITLE,FOTITLE,RMNUM>))
```

If the next line of input turns out to be a text line, it will be passed through together with an IMAP containing current values of the parameters, and already underlined if necessary. If the input is a command this usually causes a change in one of the parameters, and sometimes produces a command as intermediate result.

```
--- pgafmt(FIFLAG,PGNUM,T5,[],CMD,CAR,LINL,<PLNUM,INNUM,TINUM,
         CENUM,ULNUM,LSNUM,FIFLAG,HETITLE,FOTITLE,RMNUM>) <=

    cond(CMD=UNKNOWN,
         pgfmt(FIFLAG,PGNUM,T5,[],LINL,<PLNUM,...,RMNUM>),
    cond(CMD=RM,
         pgfmt(FIFLAG,PGNUM,T5,[],LINL,<PLNUM,...,FOTITLE,
                     sset(RMNUM,CAR,PAGEWIDTH,1+PAGEWIDTH)>),
       .
       .
       .
    cond(CMD=PL,
         pgfmt(FIFLAG,PGNUM,T5,nil,LINL,<sset(PLNUM,CAR,PAGELEN,
                1+HDRLENGTH+FTRLENGTH,PAGELEN),INNUM,...,RMNUM>),
         pgfmt(FIFLAG,PGNUM,T5,[CMD.CAR],LINL,<PLNUM,...,RMNUM>)
     )...))
```

The above equation is one of the cases statements represented as a deeply nested conditional.

These equations illustrate the nature of the final program. My confidence in its correctness comes from the knowledge that it is obtained by transformation from the protoprogram. Although the

transformation has been hard to carry out and serious difficulties have arisen, it has succeeded in converting the very naive protoprogram into a version approaching the efficiency of a conventional formatter. There remains the stage of converting into an imperative language, but even before this conversion the behaviour of my final NPL program is similar to that of Kernighan and Plauger's FORMAT program. Each processes successive lines of input only when they are needed, and output lines are formed as soon as possible. The manner of operation of each program is therefore a single pass through the input, producing output as it proceeds. The NPL program contains some minor inefficiencies through occasionally setting values of parameters which are immediately interrogated to cause some effect, rather than simply triggering that effect immediately, but these do not significantly degrade the program's performance.

CHAPTER 7

IMPLEMENTATION

In this chapter I describe the implementation of the ZAP transformation system, detailing what my additions to the underlying programs of Burstall and Darlington have been to achieve this implementation. I also describe the purposes of the non-trivial algorithms within my implementation.

7.1 GENERAL DETAILS

The programs I use are implemented in POP2 (Burstall, Collins and Popplestone [1977]) on a Dec10 machine. They are all experimental in nature - i.e. have been developed over a period of time during research, and are not intended to be, nor are they, the most efficient rendering of the algorithms in use.

The overall system divides into two levels; The lower level is the NPL parser and interpreter. The upper level is the transformation portion.

7.2 NPL LEVEL

The code in this level has been written by Rod Burstall. It can function on its own, used to execute NPL programs. Its size is 11K (and the POP2 compiler below this is a further 19K). This level provides the data structures for expressions which the transformation level manipulates.

NPL equations are kept in core, which for large programs can be expensive in consumption of space. In order to tackle transformation of large programs I have modified Burstall's code to cause it to store the equations in a disc file. Then when a context is created in my system, only the equations appropriate to that context are brought into core, and when new equations are formed, these are written out onto the disc file. The tradeoff is between the small amount of extra cpu time required to do the disc transfers, and the large amount of space saved by keeping the equations out of core.

Historically, NPL began as a language to conveniently express programs in a form suitable for applying the unfold/fold techniques to, but from that beginning, Burstall developed it into a language to demonstrate how clear, well constructed programs could not only be transformed, but also be more readily comprehended and verified.

7.3 TRANSFORMATION LEVEL

The upper level is the transformation portion, its size is 36K. This can be considered to consist of the following sections:

Utility section - generally useful functions used throughout the transformation level. Includes functions to manipulate expressions,

provide basic I/O between files, terminal and system, etc. Some of these functions are inherited from Darlington's system. In general, however, the bulk of Darlington's system has not been retained. His overall control structure for transformation and his functions for guiding it are entirely omitted.

Control section - interprets the ZAP control language described in Chapter 4. The details of my particular implementation are not significant. From the Users' Guide and Manual it would be possible to construct an interpreter with the appropriate behaviour.

Transformation step section - performs the actual step of transformation, involving expanding the pattern and expression, matching and building up the answer.

Default section - provides default patterns and type information.

I consider these last two sections in more detail:

7.3.1 Transformation Step Section

As explained in earlier chapters, the fundamental transformation step within my system involves supplying a pattern which indicates the approximate form of the desired answer. The expression to be transformed, and the provided pattern are each expanded, and matched. Bindings from the match (if it has been successful) are used to instantiate variables in the original unexpanded pattern to give the answer. Thus the process can be considered in three stages:

1. Expansion of expression and pattern

2. Matching of expression and pattern

3. Instantiating pattern to form answer

7.3.1.1 Expansion Of Expression And Pattern -

Expansion takes place within a given context of equations and lemmas. Expansion involves unfolding as much as possible by applying equations and lemmas which are rewrite rules. Some forms of lemmas - identity and commutative declarations - will have caused additions to the equations and rewrite rules of the context. Basic reductions in the form of rewrite rules are always provided by the system for COND (the conditional function), AND and OR (logical functions). In addition to expansion by unfolding, expressions are normalised by applying special purpose reductions to deal with iterative constructs, conditionals and where constructs.

In practice a major portion of cpu time used by my system is consumed by unfolding and normalising expressions. This is despite the context mechanism limiting attention to only the relevant equations. With my approach to transformation this expenditure of time seems unavoidable.

7.3.1.1.1 Normalisation Of Iterative Constructs -

Iterative constructs are ones using

set constructors, e.g. <: f(x) : x in S & p(x) :>

"all", e.g. ALL x in S : p(x)

or "exists", e.g. EXISTS x in S : p(x)

Darlington developed and implemented a useful set of reductions to apply to these constructs. I have incorporated his coding of

these within my system. For a full description see Darlington [1977]. A few examples will illustrate some actions of his reductions:

e.g. in the preceeding examples, suppose S is the nilset, i.e. the empty set. Then

```
<: f(x) : x in nilset & p(x) :> reduces to nilset

ALL x in nilset : p(x)   reduces to true

EXISTS x in nilset : p(x)   reduces to false
```

e.g. if S is the union of two sets, S1 union S2,

```
<: f(x) : x in S1 union S2 & p(x) :>   reduces to
  <: f(x) : x in S1 & p(x) :> union <: f(x) : x in S2 & p(x) :>

ALL x in S1 union S2: p(x)   reduces to
  (ALL x in S1 : p(x)) and (ALL x in S2 : p(x))

EXISTS x in S1 union S2 : p(x)   reduces to
  (EXISTS x in S1 : p(x)) or (EXISTS x in S2 : p(x))
```

7.3.1.1.2 Normalisation Of Conditionals -

Within NPL conditionals may be introduced either through the use of the cond function, or through the if/ifnot clauses. My system converts if/ifnot clauses to applications of cond, so that the following normalistations always apply.

Firstly, whenever possible, the cond function is moved outside of all other functions.

e.g. 2 + square(cond(T,1,3))   becomes

```
cond(T,2+square(1),2+square(3))
```

e.g. cond(T,1,3) + cond(T1,2,4)   becomes

```
cond(T,cond(T1,1+2,1+4),cond(T1,3+2,3+4))
```

Secondly, if nested conditionals depend on identical conditions, the inner such conditionals are simplified accordingly.

e.g. cond(T,cond(T,1+2,1+4),cond(T,3+2,3+4)) simplifies to

    cond(T,1+2,3+4)

Thirdly, conditionals whose conditions are of the form

variable = expression

are simplified by replacing all occurrences of the variable within the true branch of the conditional by the expression.

e.g. cond(N=1 , N+2 , N+4) becomes cond(N=1 , 1+2 , N+4)

7.3.1.1.3 Normalisation Of Where Constructs -

Where constructs are of the form

expression1 where <variable1,...,variableN>

    == expression2

variable1,...,variableN are the bound variables of the construct. expression2 must have the same type as <variable1,...variableN>, i.e. be an N-tuple whose components have types the same as those of variable1,...,variableN.

Firstly, where expressions are pushed inside other constructs until their left hand sides are variables or other where constructs.

e.g. (1+square(N)) where <N> == <2> becomes

    1+square(N where <N> == <2>)

In doing so, redundant where constructs are removed

e.g. (N + M) where <N> == <2> becomes

    (N where <N> == <2>) + M

Secondly, if the expression to the right of the "==" is in the form of a tuple, the where clause may be simplified by substituting the

corresponding expressions in place of the bound variables.

e.g. N where <N> == <2> simplifies to 2

e.g. M where <M> == (<N> where <N,P> == f(X)) simplifies to

N where <N,P> == f(X)

7.3.1.2 Matching Of Expression And Pattern -

The basis of my transformation step is a match between expanded pattern and expression. The match is 2nd order, since variables within the pattern are to match to functions and constructs. For an excellent account of 2nd order matching, see Huet and Lang [1977]. My matcher differs from a conventional 2nd order matcher by being somewhat restricted in some respects, and extended in others.

The restrictions are straightforward - function variables are not permitted to match to certain of the functions and constructs within the expression. Those function variables arising from $$'s within the original pattern are inhibited from matching to iterative constructs, and functions which are either restricted or have equations within the current context but not declared usable. Those function variables arising from &&'s within the original pattern are inhibited from matching to functions with equations in the current context but not declared usable.

The first extension to conventional matching concerns function variables arising from &&'s within the original pattern. Such variables are intended to create definitions of new functions, the bindings formed during the match providing these definitions. A conventional binding would be of the form

lambda x y ... z . <expression>

corresponding to an equation for the new function, newf say,

newf(x,y,...,z) <= <expression>

Note in particular that function newf is the only function occurring in the left hand side of its defining equation. The left hand side could be more complex however: The arguments of newf may contain calls to other functions, e.g.

newf(g(x),y,...,z) <= <expression>

or newf itself may be within the argument of some function, e.g.

g(newf(x,y,...,z)) <= <expression>

In the latter case my system will postulate inverses to adjust the definition of newf to bring newf to the outside of the left hand side, e.g.

newf(x,y,...,z) <= ginv(<expression>)

where ginv is the inverse of g. This restructuring is only going to be possible if the surrounding functions are unary.

My extension of 2nd order matching is to permit matches of this form, giving rise to the more complex forms of definitions for the new function. Note that in general such definitions are not executable within NPL. Darlington terms such equational definitions "implicit equations". The intention is to later use the system to transform such implicit definitions into conventional recursive equations.

The second extension to conventional matching is to match up to associativity and/or commuativity when some of the functions within the expression or pattern are declared to have these properties. In this respect I follow Topor [1975].

7.3.1.3 Instantiating Pattern To Form Answer -

Following a successful match between expanded pattern and expression, the system uses the bindings formed to construct the answer from the original pattern. At this stage a certain amount of further tidying up is also performed.

Bindings of $$ function variables are used to instantiate the original pattern, eliminating all $$'s therein.

e.g. pattern : $$(N,M)

binding for $$: lambda X Y . (X + Y) + Y

answer : (N + M) + M

Bindings for && function variables are used to create definitions of new functions. Within the original pattern, the &&'s become calls to these new functions.

e.g. pattern : &&f(succ N)

binding for f : f(X) <= X + X

answer : f(succ N) with equation

f(X) <= X + X

If the same new function was mentioned several times in the original pattern, it is possible for these separate occurrences to have led to different bindings. After the match these are put together to form a single function returning a tuple of results.

e.g. pattern : &&f(N) + f(N)

binding for f : first occurrence, f(X) <= g(X)

second occurrence, f(X) <= h(X)

answer : P + Q where <P,Q> == f(N) with equation

f(X) <= <g(X),h(X)>

Redundant arguments of new functions are automatically discarded

e.g. pattern &&f(N,M)

binding for f : f(X,Y) <= X + X

answer before simplification : f(N,M) with equation

f(X,Y) <= X+X

simplified answer : f(N) with equation

f(X) <= X+X

Some of the distributing of where expressions down branches of expressions prior to matching may need reversing in the answer and new function definitions. When the same where clause occurs in multiple branches of an expression, these occurrences are removed and a single where expression inserted at the join of these branches

e.g. f((X where <X,Y> == g(Z)) + (Y where <X,Y> == g(Z))

becomes

f(X+Y where <X,Y> == g(Z))

Although these tidying-up operations are individually trivial, the combined effect of them is to greatly ease the use of the transformation step, freeing the user from the need to perform many such trivial tasks himself.

7.3.2 Default Section

There are three default mechanisms within the system, used to generate type information, cases and patterns. I explain these, and consider how they might be extended to provide more assistance to the user.

7.3.2.1 Type Information Default --

When type information for some type is required, the default mechanism looks at the NPL DATA declaration for that type. Each of the cases on the right hand side of the declaration will form a case in the type information. If the case is in the form of a constructor with arguments, the arguments are generalized to variables, and if they are the type being defined, the generated variable is added as a "recursive case" of that case.

e.g. DATA truval <= true ++ false

is converted into

```
TYPEINFO T <= true
           <= false
```

e.g. DATA num <= 0 ++ succ(num)

gives cases 0 and succ(num). The latter has argument num, so this is generalized to a variable forming case succ(N). Since num is the type being declared, the variable is added as a recursive case, giving

```
TYPEINFO N <= 0
           <= succ(N) , N
```

e.g. DATA set(alfa) <= nilset ++ conssset(alfa,set(alfa))

is converted into

```
TYPEINFO S <= nilset
           <= consset(A,S) , S
```

When the straightforward type information that this default machanism generates is inappropriate, the user may provide his own. Typical occasions when this is necessary are when the user seeks a

recursion which restructures the data rather than simply recursing on sub-components of it. Two examples of this have appeared before in this thesis:

During transformation of the telegram problem the data type instream, instream <= in(list list char) is used. We required type information

```
TYPEINFO INS <= in(nil::CLL)
             <= in((sp::CL)::CLL) , in(CL::CLL)
             <= in((ap(A)::CL)::CLL) , in(CL::CLL)
```

The transformation of eqtips (second example of the transformation system primer) was based on restructuring binary trees, the data type

```
DATA trees(alfa) <= tip(alfa) ++ tree(trees(alfa),trees(alfa))
```

by using type information

```
TYPEINFO T <= tip(A)
           <= tree(tip(A),T) , T
           <= tree(tree(T1,T2),T) , tree(T1,tree(T2,T))
```

The former example could perhaps be generated by a slightly more sophisticated default mechanism, and such a mechanism might be a useful addition to the system. The latter example is more tricky, because the last case does not decompose at all, <u>only</u> restructures. It has been my policy to leave such trickery to the user, who has the insight to see what form is required and when.

7.3.2.2 Cases Default -

When the user prefixes an argument within the left hand side of a goal with CASESOF, this calls in the cases default mechanism to split the goal by considering the different cases that argument may take.

To do this, the mechanism examines the current type information corresponding to the type of the argument in question. The cases in the type information are taken as the cases for the argument.

e.g. GOAL funnyplus(CASESOF N,M)

with type information

TYPEINFO N <= 0 ++ succ N

produces

GOAL funnyplus(0,M)

GOAL funnyplus(succ N,M)

For parameterised types (e.g. set(alfa)) the particular instance of the type of the argument is matched to the parameterised type, and the bindings so formed used in generating variables of the appropriate types for the cases.

e.g. GOAL union(CASESOF NS,MS)

where NS has type set(num), and with type information for set(alfa)

TYPEINFO S <= nilset ++ consset(A,S)

set(alfa) is matched to set(num), binding alfa to num, and this is then used to generate from consset(A,S) consset(N,NS) where N is of type num, and NS of type set(num). Thus the goals generated are

GOAL union(nilset,MS)

GOAL union(consset(N,NS),MS)

7.3.2.3 Pattern Default -

This default generates simple patterns for use in right hand sides of goals. When an argument of the left hand side of a goal is prefixed by RECURSE, the default mechanism is called to generate a simple pattern.

The pattern formed consists of the function variable $$, around the following arguments:

all the free variables of the left hand side

recursive calls of the left hand side - recursing on the prefixed arguments.

To form the recursive calls, the prefixed argument is matched to the cases of the corresponding type information. If a match is found, and the matched case has recursive cases, these are instantiated to form the arguments for the recursive calls

e.g.
```
GOAL funnyplus(RECURSE succ succ P,Q)
together with TYPEINFO N <= 0
                          <= succ N , N
```

succ succ P is matched by succ N, binding N to succ P. Instantiating the recursive case, N, gives succ P, so the generated pattern is

```
$$(P,Q,funnyplus(succ P,Q))
```

e.g.
```
GOAL treefunction(RECURSE tree(TR1,TR2))
together with TYPEINFO T <= tip(A)
                          <= tree(T1,T2) , T1 , T2
```

generates pattern

```
$$(TR1,TR2,treefunction(TR1),treefunction(TR2))
```

As a means of generating simple patterns, this default is effective. The most obvious occasions when the patterns fail are when functions include arguments which accumulate an answer being built up.

e.g. +++ aplus(num,num) <= num

--- aplus(0,M) <= M

--- aplus(succ N,M) <= aplus(N,succ M)

Clearly aplus(N,M) = N+M, but aplus is not a primitive recursive function because its second argument is an accumulator for the answer.

When making recursive calls to such functions, the argument(s) which are accumulators will not usually remain the same in the recursive call, so the default pattern (which would leave such an argument unchanged) would fail. This is reminiscent of the difficulty Boyer and Moore's original LISP theorem prover had with such functions.

A means of adjusting the default mechanism to cope with this problem might be to put $$ around all free variables of the left hand side in all positions within recursive calls which were not being recursed upon (just in case they were accumulators).

e.g. for the earlier example of funnyplus, generate pattern

$$(P,Q,funnyplus(succ P,$$(P,Q)))

Such liberal use of $$'s in argument positions will tend to slow down matching. With a small amount of analysis, accumulators could be detected so that only their argument positions need special treatment.

CHAPTER 8

CONCLUSIONS

In this chapter I summarise the work I have done, reflect on the possible continuations I see for it, and contrast the overall approach with that of other researchers.

8.1 SUMMARY

The motivation for my research has been the need for better methods of developing software. The method I have concentrated on is transformation, and I have taken a particular transformation technique invented by Darlington and Burstall, and investigated its application to larger examples. If transformation is ever to be a practical method, a transformation system to help us is essential, and a large portion of my effort has been devoted to developing such a system.

The system and techniques are arranged in a hierarchical structure. The very lowest level of this consists of the small manipulations, folding, unfolding etc., which act as the foundation of the whole structure. The repeated application of these small manipulations would suffice to carry out our transformations, but on all except the most trivial programs they are too small scale to be practical.

Each successive level of the hierarchy serves to provide a higher-level view of transformations, whose justification lies in expansion to the next level down.

The level above fold/unfold is that of patterns - these are the primary means of guidance for individual transformation steps within my system. The user need hardly ever consider the details of the lower level folding and unfolding, and patterns are the lowest level of guidance he gives. Surrounding the use of patterns is the system's control language for setting up the context in which transformations take place. The basic commands can be viewed as the operations within this level of the hierarchy. The system provided defaults can now be seen to act as means of simplifying the generation of sequences of these operations.

Moving to the next level of the hierarchy, we find the tactics for making efficiency improvements to programs. At present these serve as entirely hand-applied aids to transformations. The application of a tactic will require the user to provide a sequence of system commands to implement that tactic.

Finally the highest level of the hierarchy is the overall strategy the user follows in performing the transformation. A strategy will expand into a sequence of applications of tactics - again this is entirely hand performed at present. Nevertheless, the user benefits be being able to see the overall organisation of the transformation if he adopts such a strategy.

The questions that now need to be asked are

Have my techniques and system been adequate for the problems considered ?

What is the range of applicabilty of this approach ?

8.1.1 Adequacy Of Techniques And System

The transformations of the telegram problem and the simple compiler have been achieved using the system and techniques developed. The final NPL programs produced are as near as we can get in NPL to the efficient iterative solution, and the changes in structure and efficiency between start and end are very major.

The last stage of converting from NPL to some imperative language is obviously crucial to the success of this approach. Since NPL has no destructive operations, there is an inherent inefficiency in NPL programs. I recognise that conversion to an imperative language is a non-trivial step, and certainly an area for further investigation. My concern has been in the transformations before this last step.

In tackling the text formatter the system was barely adequate, and it became clear that there are some aspects of the large transformation that are not being adequately captured in any level of the hierarchical organisation. At the tactics level the combination of two complex functions may give rise to a choice between alternative calling structures of the combination, and what appears to be missing is the ability to express that choice in anything but the specific system commands of the next level down.

The need to diverge at one point from the overall improvement strategy should not be regarded as a failure, indeed it is encouraging that so much of the transformation was achieved following the simple strategy, and by considering the hierarchy we see that the

simple strategy is behaving as a default to suggest a sequence of tactics to be applied, from which we diverge only if our intuition tells us it is necessary to do so.

Problems <u>have</u> been brought to light by the text formatter transformation. This exercise has not had purely negative results - possible means to overcome the new problems have became apparent. One such improvement has been the incorporation of code to cause equations to be stored in a disc file, and only brought into core when required within the context of a transformation. This enabled the text formatter to be attempted within the available computing resources.

Above all, this work has reinforced the need to always try larger examples rather than simply assume that current techniques will suffice and that no new difficulties will arise.

Program maintenance is an area I have not had time to explore in depth, but certainly one of great practical importance. From the small experiments I have tried, the results are encouraging. The structured transformations do help when it comes to modification. Again there is a need for more investigation by trying larger examples to see what difficulties occur.

8.1.2 Range Of Applicabilty

A serious question is what is the range of programs that this approach can be applied to. In terms of sheer size of the problems, it is clear that this approach will need further development in order to tackle transformations any larger than that of the text formatter.

The fact that it has been used upon a program of that size, much larger than the trivial examples upon which the methods were originally developed, suggests that the approach is not without merit. As for the width of the range, it is clear that the declarative nature of NPL is unsuitable for representing programs which rely upon sophisticated side effects, for example list copying algorithms that achieve their time and space efficiency by cunning manipulation of pointers within data structures. A good deal of the effort of programming is not concerned with such problems, rather with the overall organisation of large programs. Certainly there will always be the need to make some portion as efficient as possible, and if necessary such portions will have to be individually optimised (I do not preclude the use of other transformation techniques to achieve this). I argue that the program organisation _can_ usually be tackled by the transformational approach.

8.2 EXTENSIONS

8.2.1 System Improvements

The first class of extensions are those which can be seen as obvious improvements of the existing system.

The simplest improvements would be upgrades rather than extensions - improving efficiency of the system, and interaction with the user. Such improvements tend to be never ending, as continued use of the system highlights the areas most needing attention.

Within chapter 7, Implementation, I suggested how the default mechanisms could be enhanced to do more for us. I have been wary of

incorporating too many defaults into the system, preferring to let the user guide the system through non-trivial transformations, and introducing a default only when I perceive it to be generally useful. Investigation of more examples is required to see which of the possible extensions to defaults are truly useful.

At present the transformation strategy and tactics are human-generated, and serve to help the user guide the system, rather than guide the system directly. The "combine" and "tuple" tactics could be incorporated as commands to the system, which would be expanded into sequences of conventional transformation commands (CONTEXT USING ...). In the same fashion as the provision of defaults for patterns etc., I envisage a default being used to expand tactics into commands for simple examples, with the user stepping in only for more complex problems. From the hierarchical viewpoint we see that such an extension would essentially be the automation of the tactics level. In a similar manner the strategy level could be included within the system, and here the simple strategy I have been following would be provided as a default at this level.

8.2.2 Extending Transformation Methods

As I have already discussed in the summary, the transformations I deal with lead to programs within NPL, and conversion to an imperative language is still required. This suggests that the transformation system may form only part of a larger program development system, where a final NPL program is input to some further stage to do the conversion. This is more a matter of plugging in the system into a larger machine than extending it. More

interesting are the possibilities for extending the transformation methods themselves:

One such extension arises from taking heed of schemata-driven program transformation. (Darlington and Burstall [1976]) Transformations within my system very often do not rely upon the detailed behaviour of all the functions involved, for some of the functions perhaps only a few of their properties are significant. Thus when faced with the transformation of similar programs, whose functions are identical with respect to these properties, essentially the same transformation will suffice. This suggests that the first transformation could be generalized to form a transformation schema, the input being the generalized initial program, the output the generalized transformed program. Then when given what looks like a similar problem, it could be matched against the schema input and if successful, the schema output would be instantiated to give the answer. This would require the introduction of a schema matcher and some procedure for generalizing from a specific transformation to a schema. An alternative approach would be to transform schematic programs directly. This can be viewed as verifying schemata in terms of the unfold/fold operations, thus having the advantage of providing an easily extendible and verifiable set of schemata. Interestingly, whilst the possibility of incorporating scehmata was apparent before the transformation of the text formatter, it was only during this that new difficulties arose to suggest that schemata might be necessary. I see their incorporation as the logical next step in the development of the system.

Another direction to consider is based on changing NPL in some manner. Two deficiencies of NPL are the lack of any suitable form of

data abstraction, and of higher order functions. Their inclusion would add to our tools for writing well modularised easy to comprehend programs. At Edinburgh Burstall and MacQueen are working on a new language, HOPE, which is essentially NPL with these features included. The transformation methods will require extension to cope with them. Hopefully such an extension would be fairly natural. Much more radical a change would be to introduce some form of destructive operator into NPL. The extension of the transformation techniques to cope with this would be much harder, however the potential benefits - the ability to transform to imperative programs and investigate algorithms relying upon destructive operations - make this an enticing area for study.

8.3 COMPARISON WITH OTHER WORK

Finally I contrast this approach to transformation with the work of other researchers.

One of the key decisions underlying my approach is the acceptance of user guidance. This means that my system does not attempt to transform programs totally automatically as does Manna and Waldinger's DEDALUS system. As a consequence of this I am able to tackle very much larger transformations which would be beyond the capabilities of DEDALUS like systems. The problems that automatic systems can handle (which might occur within larger transformations) require a small amount of user guidance within my system, and my approach has been to incorporate a few default mechanisms which the user can call into action when he perceives a transformation to be straightforward.

Interactive systems which accept a small amount of guidance from the user, e.g. Darlington's system based upon folding/unfolding, are able to achieve more than the entirely automatic systems, but are themselves incapable of tackling the large transformations I have been considering. The semi-automatic systems commonly incorporate several "strategies" for performing entire transformations which the user switches on or off. Within my approach such "strategies" are replaced by defaults within the hierarchical levels of my system. Instead of being limited to only switching them on or off, the user can apply them and override them at the points where his intuition tells him to do so. If automatic or semi-automatic systems are to be developed to be applicable to larger programs, I feel that some form of structuring of the transformation process akin to my hierarchical arrangement is essential.

The scope of my system is limited to manipulations within recursion equations, hence I am unable to tackle problems such as recursion removal and conversion to imperative languages, which systems such as Darlington and Burstall's schema based work was developed for. Bauer's proposed system would be very wide-ranging, encompassing the whole spectrum from high-level synthesis to manipulation of machine code. Significantly this system is planned to rely entirely upon user guidance to direct its application through a transformation.

Lastly there is the hand-performed transformation work, which has been applied to relatively complex algorithms, sometimes as a means of verification rather than software construction, as epitomised by Martelli's transformation of an algorithm to copy cyclic list structures. The final programs produced can be very

complex in operation, making use of side-effects, structure sharing, etc. This requires sophisticated reasoning during the transformation, well beyond the capabilities of any existing machine-based system. The size of the programs transformed by hand is not as great as that of the text formatter however, which suggests that although hand-transformation may be suitable for complex but compact algorithms, to transform a straightforward but long program is best done with machine aid.

BIBLIOGRAPHY

Arsac, J. [1977] La Construction to Programmes Structures. Dunod, Paris. (especially Chapters IX and XII).

Aubin, R. [1975] Some generalisation heuristics in proofs by induction. Proceedings of International Symposium on Proving and Improving Programs, Arc-et-Senans, France, pp 197-208

Aubin, R. [1976] Mechanising structural Induction. Ph.D. thesis. Dept. of Artificial Intelligence, University of Edinburgh

Bauer, F.L., Broy, M., Partsch, H., Pepper, P. and Wossner, H. [1978] Systematics of transformation rules. TUM-INF-BER-77-12-0350 Institut fur Informatik, Technische Universitat Munchen.

Bauer, F.L., Partsch, H., Pepper, P. and Wossner, H. [1977] Notes on the project CIP: outline of a transformation system. TUM-INFO-7729 Institut fur Informatik, Technische Universitat Munchen.

Broy, M. [1977] Program development : the Ackermann function as an example. TUM-INFO-7716 Institut fur Informatik, Technische Universitat Munchen.

Broy, M. [1978] A case study in program development : sorting. TUM-INFO-7831 Institut fur Informatik, Technische Universitat Munchen.

Boyer, R. S. and Moore, J S. [1973] Proving theorems about LISP functions. Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford, California, pp 486-493

Burstall, R.M. [1969] Proving properties of programs by structural induction. Computer Journal vol 12 no. 1 pp 41-48

Burstall, R.M. [1977] Design considerations for a functional programming language. Proc. of Infotech State of the Art Conference, Copenhagen, pp 45-57

Burstall, R.M., Collins, J.S. and Popplestone, R.J. [1977] Programming in POP2. University Press, Edinburgh.

Burstall, R.M. and Darlington, J. [1975] Some transformations for developing recursive programs. Proceedings of International Conference on Reliable Software, Los Angeles, California, pp 465-472.

Burstall, R.M. and Darlington, J. [1977] A transformation system for developing recursive programs. JACM vol 24 no. 1 pp 44-67 (revised and extended version of their Los Angeles paper, 1975).

Burstall, R.M. and Feather, M.S. [1978] Program development by transformation: an overview. Proceedings of Toulouse CREST Course on Programming, Toulouse.

Chatelin, P. [1976] Manipulation de programmes: quelques Transformations par duplication de boucles. Expose aux Journees Informatiques de Nice, Universite de Nice.

Chatelin, P. [1977] Self-redefinition as a program manipulation strategy. In Proceedings of Symposium on Artificial Intelligence and Programming Languages, ACM SIGPLAN NOTICES and SIGART NEWSLETTER Aug 77 pp 174-179

Clark, K. and Darlington, J. [1977] Algorithm classification through synthesis. Internal Report, Dept. of Computing and Control, Imperial College, London. To appear in Computer Journal.

Cohn, A. [1979] High level proofs in LCF. In Proceedings of the 4th International Workshop on Automated Deduction, Austin, Texas, February 1979.

Cousineau, G. [1976] Un systeme complet pour l'equivalence des schemas iteratifs, 2me Colloque international de l'Institut de Programmation.

Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. [1972] Structured Programming. Academic Press.

Darlington, J. [1972] A semantic approach to automatic program improvement. Ph.D. thesis. Dept. of Machine Intelligence, University of Edinburgh.

Darlington, J. [1975] Applications of program transformation to program synthesis. Proceedings of International Symposium on Proving and Improving Programs, Arc-et-Senans, France, pp 133-144

Darlington, J. [1976] Transforming specifications into efficient Programs. Invited paper at IFIP Working Group 2.1 Conference on Software Specifications, St. Pierre-de-Chatreuse, France. Also published in New Directions in Algorithmic Languages.

Darlington, J. [1976a] A synthesis of several sorting algorithms. D.A.I. Research Report no. 23, Dept. of Artificial Intelligence, University of Edinburgh. To appear in Acta Informatica.

Darlington, J. [1977] Program transformation and synthesis: present capabilities. D.A.I. Research Report no. 48, Dept. of Artificial Intelligence, University of Edinburgh. Also Research Report no. 77/43, Dept. of Computing and Control, Imperial College of Science and Technology, London.

Darlington, J. [1978] Program transformation involving unfree data structures an extended example. Proceedings, 3eme Colloque International sur la Programmation, Paris.

Darlington, J. and Burstall, R.M. [1973] A system which automatically improves programs. Proceedings of Third International Joint Conference on Artificial Intelligence, Stanford, pp 479-485

Darlington, J. and Burstall, R.M. [1976] A system which automatically improves programs. Acta Informatica 6, pp 41-60 (First appeared as their [1973] paper).

Dijkstra, E.W. [1976] A Discipline of Programming. Prentice Hall.

Feather, M.S. [1978] Program transformation applied to the telegram problem. Proceedings, 3eme Colloque International sur la Programmation, Paris.

Feather, M.S. [1978a] "ZAP" program transformation system primer and users' manual. D.A.I. Research Report no. 54, Dept. of Artificial Intelligence, University of Edinburgh.

Floyd, R.W. [1967] Assigning meanings to programs. Proceedings of Symposium in Applied Mathematics, American Mathematical Society, vol 19 pp 19-32

Gerhart, S.L., Lee, S. and deRoever, W.P. [1979] The evolution of list copying algorithms. Proceedings, 6th ACM POPL Symposium, Texas. pp 53-67

Gnatz, R. [1977] Zur konstruction von programmen durch transformation. TUM-INFO-7741 Institut fur Informatik, Technische Universitat Munchen.

Gnatz, R. and Pepper, P. [1977] fusc: An example in program development. TUM-INFO-7711 Institut fur Informatik, Technische Universitat Munchen.

Good, D.I. [1970] Toward a man-machine system for proving program correctness. Ph.D. Thesis, University of Wisconsin.

Green, C. [1976] The design of the PSI program synthesis system. Proc. of the Second International Conference on Software Engineering, San Francisco, California, pp 4-18.

Green, C. and Barstow, D. [1975] Some rules for the automatic synthesis of programs. Advance Papers of the Fourth International Joint Conference on Artificial Intelligence. Tbilisi, Georgia, USSR, pp 232-239.

Green, C. and Barstow, D. [1977] Program synthesis knowledge for efficient sorting. Draft report, A.I. Lab, Computer Science Dept., Stanford University, California.

Gordon, M., Milner, R. and Wadsworth, C. [1976] The LCF manual. Dept. of Computer Science, University of Edinburgh.

Hayes, P.J. [1973] Computation and deduction. Proc. of 1973 Mathematical Foundations of Computer Science, Czechoslovakian Academy of Sciences.

Henderson, P. and Morrison, J. [1976] A lazy evaluator. 3rd Symp. on Principles of Programming Languages, Atlanta. pp 95-103

Henderson, P. and Snowdon, R. [1972] An experiment in structured programming, BIT 12 pp 38-53

Hoare, C.A.R. [1969] An axiomatic basis for computer programming. CACM vol 12 no. 10 pp 576-583

Huet, G. and Lang, B. [1977] Proving and applying program transformations expressed with second-order patterns. IRIA Rapport de Recherche no. 226

Jackson, M. [1975] Principles of Program Design. Academic Press.

Jones, C.B. [1976] Program development using data abstraction. IBM ESRI, La Hulpe, Belgium.

Kernighan, B.W. and Plauger, P.J. [1976] Software Tools. Addison-Wesley.

Kibler, D.F. [1978] Power, efficiency, and correctness of transformation systems. Ph. D. Thesis, University of California.

Kibler, D.F., Neighbors, J.M. and Standish, T.A. [1977] Program Manipulation via an efficient production system. In Proceedings of Symposium on Artificial Intelligence and Programming Languages. ACM SIGPLAN NOTICES and SIGART NEWSLETTER Aug 77 pp 163-173

Kott, L. [1978] About transformation system : a theoretical study. Proceedings, 3eme Colloque International sur la Programmation, Paris.

Kowalski, R. [1977] Programming = logic + control. Imperial College Report.

Kreig-Bruckner, B. [1978] Concrete and abstarct specification, modularisation and program development by transformation. TUM-INFO-7805 Institut fur Informatik, Technische Universitat Munchen.

Ledgard, H. [1974] The case for structured programming. BIT 14 pp 45-57

Lovemann, D. [1977] program improvement by source to source transformation. JACM vol 12 no. 1 pp 121 - 145

Manna, Z. [1969] The correctness of programs. Journal of Computer and System Sciences, vol 3 no. 2 pp 119-127

Manna, Z. and Waldinger, R.J. [1975] Knowledge and reasoning in program synthesis. Artificial Intelligence vol 6 no. 2 pp 175-208

Manna, Z. and Waldinger, R.J. [1977] The automatic synthesis of recursive programs. Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages pp 29-36

Manna, Z. and Waldinger, R. [1977a] Synthesis: Dreams => Programs. Stanford AI Memo 302, Stanford University.

Martelli, A. [1978] Program development through successive transformations: an application to list processing. Proceedings, 3eme Colloque International sur la Programmation, Paris.

McCarthy, J. [1963] A basis for a mathematical theory of computation. Computer programming and formal systems pp 33-70, Edited by P.Braffort and D.Hirshberg, North Holland, Amsterdam.

McKeeman, W.M. [1976] Respecifying the telegram problem. TR-77-2-001 Information Sciences, University of California, Santa Cruz.

Milner, R. [1972] Logic for Computable Functions; Description of a Machine Implementation. AI Memo no. AIM-169, Computer Science Dept., Stanford

Moore, J S. [1974] Introducing Iteration into the Pure LISP Theorem Prover, CSL-74-3, Xerox Palo Alto Research Center, Palo Alto, California.

Partsch, H. and Pepper, P. [1976] A family of rules for recursion removal related to the "Towers of Hanoi" problem. Rep. no. 7612 Institut fur Informatik, Technische Universitat Munchen.

Partsch, H. and Pepper, P. [1977] Program transformations on different levels of programming. TUM-INFO-7715 Institut fur Informatik, Technische Universitat Munchen.

Pettorossi, A. [1977] Transformation of programs and use of "tupling strategy". Proceedings of Informatica 77 Conference, Bled, Yugoslavia.

Pettorossi, A. [1978] Improving memory utilization in transforming programs.

Schmitz, L. [1978] An exercise in program synthesis : algorithms for computing the transitive closure of a relation. Bericht no. 7801, Fachbereich Informatik, Hochschule der Bundeswehr Munchen.

Schwarz, J. [1977] Using annotations to make recursion equations behave. D.A.I. Research Report no. 43, Dept. of Artificial Intelligence, University of Edinburgh. Also appeared in IEEE transactions on Software Engineering.

Schwarz, J. [1978] Verifying the safe use of destructive operations in applicative programs. Proceedings, 3eme Colloque International sur la Programmation, Paris.

Sickel, S. [1977] A logic-based programming methodology. Technical Report no. 77-8-001, Information Sciences, University of California, Santa Cruz, California.

Sickel, S. [1978] Removing redundant recursion. Technical Report no. 78-8-003, Information Sciences, University of California, Santa Cruz, California.

Standish, T., Harriman, D.C., Kibler, D.F. and Neighbors, J.M. [1976] The Irvine program transformation catalogue. Dept. of Information and Computer Science, Univ. of Cal. Irvine, Irvine, Cal.

Standish, T., Harriman, D.C., Kibler, D.F. and Neighbors, J.M. [1976a] Improving and refining programs by program manipulation. Proc. ACM Annual Conference, pp 509-516

Topor, R. [1975] Interactive program verification using virtual programs. Ph.D. thesis, University of Edinburgh.

Waldinger, R.J. [1977] Achieving several goals simultaneously. Machine Intelligence 8: Machine Representations of Knowledge (Eds: E.W.Elcock and D.Michie) Ellis Horwood Ltd., Chichester, England.

Warren, D. [1977] Implementing Prolog - compiling predicate logic programs. D.A.I. Report no. 39,. Dept. of Artificial Intelligence, University of Edinburgh.

Wegbreit, B. [1976] Goal-directed program transformation. IEEE Transactions on Software Engineering, vol SE-2, no. 2 pp 69-80.

Wirth, N. [1971] Program development by stepwise refinement. CACM vol 14 no. 4 pp 221-227

Wirth, N. [1973] Systematic Programming. Prentice Hall.

Zahn, C.T. [1976] A brief analysis of the telegram problem. Unpublished

APPENDIX A

NPL

NPL is a first order recursion-equation language designed and implemented by Rod Burstall at Edinburgh based on work by him and J. Darlington. Burstall [1977] provides a brief descirption of NPL, motivations behind its design, and desirable extensions to it. In this appendix I informally describe the version of the language used in the ZAP transformation system.

An NPL program consists of

infix and prefix declarations - these declare symbols to be infixes or prefixes

data definitions - by which the programmer introduces his own data types

type declarations for functions and variables

recursion equations for functions

A program is surrounded by DEF...END. Once the appropriate definitions have been made, the user evaluates expressions by enclosing them within VAL...END. The answer is printed on the terminal. NPL at present makes no distinction between upper and lower case, but for clarity I adopt the convention of upper case for

NPL keywords and variables, lower case for everything else.

A.1 INFIX AND PREFIX DECLARATIONS

Symbols to be used as infixes or prefixes must first be declared as such.

e.g. INF 4 + - declares + and - to be infixes with precedence 4 (to indicate how tightly to bind to its argument when expressions are not fully parenthesised).

e.g. PRE 20 succ declares succ to be a prefix with precedence 20 thus succ N + M parses as succ(N) + M rather than succ(N + M)

A.2 DATA DEFINITIONS

The user may define his own data types by means of data definitions.

e.g. DATA weekday <= mon ++ tue ++ wed ++ thu ++ fri

DATA is a keyword to the NPL interpreter, announcing a data definition. "<=" and "++" are special symbols too; to the left of the "<=" is the name of the data type being defined; to the right, separated by "++"'s, are the cases of the defined type.

Thus the example is defining a new type, weekday, which has 5 different cases, mon tue wed thu fri.

Similarly we might say

DATA truval <= true ++ false (i.e. truthvalues)

DATA weekend <= sat ++ sun

Previously defined data types may occur in definitions of new types

e.g. DATA day <= dy(weekday) ++ dy(weekend)

The "dy" is a constructor for days, converting a weekday or weekend into a day. This is necessary;

DATA day <= weekday ++ weekend

would not suffice, since then "mon" could be either a weekday or a day.

More interesting data types are built up recursively

e.g. PRE 20 succ

DATA num <= 0 ++ succ num

defines a type num, either 0 or succ num. Thus expressions of this type are 0, succ 0, succ succ 0, etc. This type represents natural numbers, either 0 or the application of succ (short for successor) to a number. Thus 5 would be written succ succ succ succ succ 0.

Data types may be parameterised.

e.g. INF 4 ::

DATA list(alfa) <= nil ++ alfa::list(alfa)

defines a parameterised type list, of alfa"'s. "::" is an infix constructor for lists, i.e. the infix form of cons in the LISP notation. The parameterisation allows us to build lists of any type we like (e.g. list(num), list(day), list(list(num)) etc).

alfa acts as a type variable and is predefined by the NPL system (other predefined type variables are beta,gamma, delta, epsilon).

Data types may be mutually recursive

e.g. DATA glob1 <= nil1 ++ g1(glob2);

glob2 <= nil2 ++ g2(glob1)

the ";" separates the data definitions to the right of the single "DATA", allowing the reference to glob2 prior to its definition.

Some data types are already declared by the NPL system. These are truval, num, list(alfa) and set(alfa). (This last defined by DATA set(alfa) <= nilset ++ consset(alfa,set alfa))

Numbers may be typed in as decimals rather than many succ's. The interpreter converts them to the succ form for evaluation, and prints them out as decimals afterwards.

Lists may be input without needing to use many "::"'s by putting the elements to go into a list within square brackets, separated by commas.

e.g. [1,2,3] is equivalent to 1::(2::(3::nil)))

The interpreter reads in lists in either form, and prints them out using square brackets.

A.3 TYPE DECLARATIONS FOR FUNCTIONS AND VARIABLES

All symbols to be used as functions or variables must have their type declarations made in advance.

Variable type declarations take the form

VAR variable names separated by commas : type

VAR is a special symbol to the interpreter, announcing the start of a type declaration.

e.g. VAR N,M : num declares N and M to be variables of type num

e.g. VAR NL : list(num) declares NL to be of type list(num)

e.g. VAR A,A1 : alfa declares A and A1 to be of type alfa

e.g. VAR omega : type declares omega to be a type variable

Function declarations take the form of an equation, the left hand side being the function symbol applied to argument(s) which are its input type(s), the right hand side the result type.

e.g. +++ square(num) <= num

"+++" is another special symbol to the interpreter, indicating the following equation is a function declaration. Thus in the example square is declared to be a unary function accepting an argument of type num, and producing a result of type num.

e.g. INF 6 +

    +++ num + num <= num

declares + to be an infix binary function, taking numxnum to num.

e.g. +++ length(list(alfa)) <= num

makes type declaration for length. Note the use of a parameterised type.

A.4 RECURSION EQUATIONS FOR FUNCTIONS

These take the form

--- left hand expression <= right hand expression

"---" is yet another special symbol to the interpreter. The left hand expression is of the form f(e1,...,en), n >= 0, where f is the function being defined, and e1,...en are expressions including only variables and constructor symbols. The right hand side is any expression, provided that all its free variables occur in the left hand expression. Before giving all the forms an expression may take, here are some simple examples:

--- length(nil) <= 0

--- length(A::AL) <= succ length(L)

```
--- square(N) <= N*N
```

Evaluation of expressions makes use of these equations. Given an expression to evaluate, the interpreter attempts to match it (or portions of it) to left hand sides of equations. If a successful match is found, the bindings of the variables in the left hand expression are used to instantiate the variables of the right hand expression to give the answer.

e.g. with the above equations, evaluating length(nil) gives 0. Evaluating length(1::nil) gives succ length(nil), which in turn gives succ 0.

Thus the equations can be thought of as rewrite rules, applied from left to right. Evaluation is call-by-value, i.e. leftmost innermost portions of expressions are evaluated first. Evaluation continues as long as possible, until no further evaluation is possible.

In addition to variables and applications of functions, expressions may take the following forms:

A.4.1 N-tuples

These are written as <e1,...,en>, n >= 1.

The type of such expressions must be written as tuplen(type1,...,typen).

e.g.
```
+++ pairnum(num) <= tuple2(num,num)
--- pairnum(N) <= < N , N+1 >
```

In fact n-tuples act as constructors, and may occur within left hand sides of recursion equations.

A.4.2 Where Constructs

These take the form

expression1 where <V1,...Vn> == expression2, n >= 1.

Expression2 must have the same type as <V1,...,Vn>, i.e. an n-tuple. V1,...Vn are variables which within the scope of expression1 become bound to the corresponding components of expression2, if it can be evaluated to an n-tuple.

e.g. N+N where <N> == <1>

e.g. N+M where <N,M> == (<P,2> where <P> == <3>)

A.4.3 Conditionals

A special three argument function, cond, is provided by the system. Its type declaration is

+++ cond(truval,alfa,alfa) <= alfa

The interpreter first evaluates the first argument, and if this evaluates to true, only then evaluates the second argument to give the answer, else if to false, the third argument.

e.g. cond(N=0 , 0 , M/N) will evaluate M/N only if N=0 evaluates to false.

For certain uses cond is somewhat clumsy to write.

e.g. cond(p0(N),f0(N),cond(p1(N),f1(N),f2(N)))

this can be written more easily as

```
--- g(N) <= f0(N) if p0(N)
         <= f1(N) if p1(N)
         <= f2(N) ifnot
```

(Omitting the all encompassing final ifnot clause is possible but

not recommended!).

A.4.4 Set Expressions

These approximate traditional mathematical notation for sets.

e.g. <: 1,2,3 :>

"<:" and ":>" are set brackets. The expression builds the set of elements 1,2,3. Thus the set brackets are analogous to list brackets, except they do extra work to remove duplicate elements.

e.g. <: f(N) : N in es & p(N) :>

es is any expression evaluating to a set.

p(N) is an optional predicate (i.e. evaluates to a truval) possibly including occurrences of variable N.

f(N) is an expression, possibly including occurrences of variable N.

The result is the set of f(N)'s for all N in es such that p(N) is true. Omitting the predicate is equivalent to putting true in its place.

e.g. <: N : N in <: 1,2,3 :> :>

evaluates to <: 1,2,3 :>

e.g. <: N+N : N in <: 1,2,3 :> & N /= 1 :>

evaluates to <: 4,6 :>

Several bound variables may range through sets

```
e.g. <: f(N,M) : N in es & p(N,M) ,
                 M in es1 & p1(M) :>
```

so, e.g. <: N+M : N in <: 1,2 :> , M in <: 3,4 :> :>

evaluates to <: 4,5,6 :>

A.4.5 "all" And "exists" Expressions

In a similar manner to set expressions, these are modelled upon traditional mathematical notation.

e.g. ALL N in es : p(N)

e.g. EXISTS N in es : p(N)   with obvious meanings.

e.g. ALL N in <: 1,2,3 :> : N /= 0   evaluates to true

A.5 EXAMPLE PROGRAM

Comments may be included in NPL programs by prefixing them with "///". They are terminated by the next NPL special symbol (i.e. one of +++ --- END INF PRE DATA). The following is an example NPL program:

```
DEF
/// define numbers, addition, mutiplication and factorial

INF 20 succ
DATA num <= 0 ++ succ num

VAR N,M : num

+++ num + num <= num
--- 0 + N <= N
--- succ M + N <= succ(M + N)

INF 6 *
+++ num * num <= num
--- 0 * N <= 0
--- succ M * N <= N + M*N

+++ factorial(num) <= num
--- factorial(0) <= 1
--- factorial(succ N) <= succ N * factorial(N)

END
```

After this we could say

VAL factorial(3) END   which evaluates to 6.

A.6 NPL SYNTAX

I adopt the following syntax conventions:

Lower cases indicates a non-terminal.
"..." indicates (optional) repitition, the separator being the item to each side of the "...".

```
NPL program ::= DEF statement...statement END

statement ::= INF precedencedec |
          PRE precedencedec |
          DATA datadef ; ... ; datadef | VAR vardec |
          +++ fndec | --- receqn

precedencedec ::= num symbol , ... , symbol

datadef ::= typeexpn = typeexpn ++ ... ++ typeexpn

typeexpn ::= symbol |
          symbol ( typeexpn , ... , typeexpn ) |
          presymbol typeexpn |
          typeexpn infsymbol typeexpn

vardec ::= symbol , ... , symbol : typeexpn

fndec ::= fndeclhs <= typeexpn

fndeclhs ::= symbol |
          symbol ( typeexpn , ... , typeexpn ) |
          presymbol typeexpn |
          typeexpn infsymbol typeexpn

receqn ::= pattern <= expn |
          pattern <= expn IF expn ifclauses

ifclauses ::= empty | <= expn IF expn ifclauses | <= expn ifnot

pattern ::= symbol |
          symbol ( patexpnlist ) |
          presymbol patexpn |
          patexpn infsymbol patexpn

patexpn ::= symbol | < patexpnlist > | [ patexpnlist ] |
          symbol ( patexpnlist ) | presymbol patexpn |
          patexpn infsymbol patexpn

patexpnlist ::= empty | patexpn , ... , patexpn

expn ::= symbol | symbol ( expnlist ) |
          presymbol expn | expn infsymbol expn |
          [ expnlist ] | < expnlist > | whereexpn |
          setexpn | allexpn | existsexpn
```

```
expnlist ::= empty | expn , ... , expn

empty ::=

whereexpn ::= expn WHERE tuple == expn

setexpn ::= <: expnlist :> |
            <: expn : generator , ... , generator :>

generator ::= symbol IN expn | symbol  IN expn & expn

allexpn ::= ALL generator : expn

existsexpn ::= EXISTS generator : expn

infsymbol ::= symbol       ** But must have been
                              declared as an infix

presymbol ::= symbol       ** But must have been
                              declared as a prefix

symbol ::= alphabetic | alphabetic alphanumeric...alphanumeric |
            sign...sign

alphabetic ::= A | B | .... | Z

alphanumeric ::= alphabetic | numeral

numeral ::= 0 | 1 | .... | 9

sign ::= + | - | * | = | < | > | : | @ | $
```