A PROGRAM DESIGN LANGUAGE PROCESSOR

Andreas Martin Dinkelacker

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the Degree of Master of Science in Engineering.

JOHANNESBURG, 1988

DECLARATION

I declare that this dissertation is my own, unaided work. It is being submitted for a degree of Master of Science in Engineering at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

Andreas Martin Dinkelacker

Fifteen day of August, 1988

ACKNOWLEDGEMENTS

Acknowledgement is given to:

- The Management of Tek Logic (Pty) Limited for their support of this project.

- Dr A J Walker for his guidance and supervision over the duration of the project.

Table of Contents

1. INTRODUCTION

A Program Design Langunge is a tool applicable to the practice of software engineering.

In order to relate its use to the practice of software engineering, a perspective is given on:-

- software design
- software engineering
- typical design tools

Some background is given on current technology in compiler construction. This is directed towards those techniques relevant to building a Program Design Language Processor.

This is followed by an overview of the processors design, conclusions reached in this project, a bibliography, and an appendix containing the user manual of the program design language processor.

2. SOFTWARE DESIGN

2.1. Definition of Software Design

Software is a changeable part of a computer system. Software consists of programs and data. It provides the character of the system. It forms a major part of the system, and its design forms a major part of the design of a system.

In order to define the scope and content of software design, it is relevant to examine engineering design as a whole.

According to Asimov (1962), "Engineering design is a purposeful activity directed towards the goal of fulfilling human needs, particularly those which can be met by the technological factors of our culture".

These needs are not met by existing solutions - thus the goal exists. The achievement of the goal is dependant on the availability of known technology and practice.

Restating this definition:- Software design is the process of fulfilling needs by the creation of programs and data.

A number of alternate definitions and views of software design exists. Some examples are:-

- Design as an integral process of problem definition and solution (Peters). This highlights the interrelated and interactive nature of design, as the creation of a solution provides insight to the problem, and the problem insight directs the solution.

- "... design is not a formulatable activity. It does not lend itself to prescriptive or sequential steps that will ensure success ... It is not possible to exhaust all reasonable variations and avenues of inquiry to reach an acceptable design. One could go on essentially forever." (Peters).

- The model of design as a wicked problem (J C Jones, H W J Rittel and M M Webber, elaborated by Peters). Some properties are:

  - A wicked problem cannot be definitively stated

  - There is no rule or guideline to determine when a wicked problem is solved

- Wicked problems do not have right or wrong solutions, only good or bad ones

- A wicked problem cannot be definitively tested

-Solutions to wicked problems are too significant to be experimented with

- Neither the number of possible solutions nor the means of obtaining them is limited

- Each wicked problem is a singularity

- Wicked problems can be viewed as symptoms of other higher-level problems

The concepts of software design are covered more fully by Freeman (1984), and design representations by Peters, and Peters and Tripp (1976).

2.2. History of Software Design

In the history of software design there is a distinct watershed. The dividing line is the emergence of structured programming.

Structured programming is the writing of programs with a significant emphasis on its form, in particular with the use of fixed constructs such as sequence, selection, iteration and abstraction.

2.2.1. Before Structured Programming - First Phase

This phase represented the solution of well established functions using computers and software.

The design activity consisted of force-fitting the tried and tested algorithms onto the computers of the day. Critical constraints were often execution speed, and program and data size. No emphasis was placed on the form of the code.

The code was generally created by application specialists of a high calibre, for their sole use. The effort was generally minimal in comparison to the value of the result.

2.2.2. Before Structured Programmin. Second Phase

This phase represented the usage of computers and software to solve significant, new problems.

These problems were orders of magnitude more complex than those of the first phase, without known solutions, and requiring teams of people for extended durations.

No emphasis was placed on the form of the code, and little on general techniques and methods.

The results were a recurring sequence of embarrassing failures. This scenario is extensively examined in Brooks (1975).

"We build systems like the Wright brothers built airplanes - build the whole thing, push it off a cliff, let it crash, and start over again" (R M Graham in Naur, Randell 1969).

Once these failures were recognised as general and predictable, the stage was set for structured programming.

2.2.3. The Re-living of History

'Those who do not learn from history are doomed to re-live it.' The two phases defined above should be relegated to history, yet they are still common today.

Typical cases in which the first phase still occurs are applications recently turning to computer based solutions or where the computer capabilities are very limited. These include calculators, single chip micro-computer based random logic replacement, peripheral and mass storage controllers, and signal processors. As before, they are characterised by established algorithms, and execution speed, program and data limitations. With effort, they are successful.

Typical areas in which the second phase still occurs are outgrowths of successful first phase areas. These include more complex peripheral controllers (SNA, X25), more "powerful" hardware and operating systems (Dijkstra 1972), and emulated terminals. These products are still often failures, because of the lack of use of software design methodology by the designers.

2.2.4. The Advent of Structured Programming

Structured programming is the disciplined practice of writing programs where emphasis is laid on the form of the program.

It can be regarded as the first structured technique (Shooman 1983).

Typical form aspects are:-

- The use of a restricted number of control constructs
- The use of indentation to indicate control structure scope
- The use of meaningful symbols
- Size limits of program modules

Definitions of structured programming include:-

- "Structured programming theory deals with converting arbitrarily large and complex flowcharts into standard forms so that they can be represented by iterating and nesting a small number of basic and standard control logic structures" (Jensen 1981)

- "Structured programming is a manner of organising and coding programs that makes the programs easily understood and modified" (Jensen 1981)

- "The fundamental concept [of structured programming] is a proof of correctness" (Jensen 1981)

- "'Structured Programming' is the formulation of programs as hierarchical, nested structures of statements and objects of computation" (Wirth 1974)

2.2.5. The Basis of Structured Programming

In the IFIPS Congress of 1965, Dijkstra proposed that the GO TO statement was not necessary in programming languages, and further that the reduction of GO TO's was related to improved programs.

In 1966 Bohm and Jacopini published a historic paper. This paper was a theoretical proof of the sufficiency of two formation rules. In practice this implied that any program could be written using minimum control structures of (sequence,) selection and iteration.

In parallel, many programmers found it was also practical to write programs in this manner (i.e. without explicit transfer of control using GO TO statements). A number of experimental projects were performed, and also languages were designed, to demonstrate this point.

Finally, in a classic letter titled "GO TO Statements Considered Harmful", Dijkstra provided a strong argument for the reduction of GO TO statements (1968). He found that the quality of programs reduced as the number of GO TO's increased. "The GO TO statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program".

In practice this implied that programs should be written using only control structures such as sequence, selection and iteration.

A paper by Curtis (1980) provides a survey of tests performed on the use of GO TO's and various structured constructs.

Many further articles and papers contributed to the GO TO controversy. For example Peterson, Kasami and Tokura (1973), who state "Recently there has been considerable interest in the possibility of replacing the use of GO TO statements in programs with interactive constructs such as FOR and WHILE because the latter are more easily understood, less error prone, and more easily proved correct". Other discussions are in Shooman (1983).

Even where the language and knowledge existed, these constructs were often not applied. For example, in one survey of 100 PL/I programs of a total of 100 000 lines of code, only 11 DO - WHILE statements were used, and 20% of programs used no IF - THEN - ELSE statements (Yourdon 1979 p 124).

The practice of structured programming initially met with a measure of opposition. It was generally condemned on the basis of reduced creativity, efficiency in code size, execution speed, and its conflict with facilities provided by the language (Shooman 1983).

As a result languages were developed to be efficient within these constraints (Wirth in Wasserman 1980).

An interesting comparison can be drawn between the initial opposition of structured programming (vs unstructured programming), and the use of high level languages (vs assembly languages).

In current practice, whilst GO TO's still exist (Jensen 1981), structured programming is the accepted method.

2.2.6. The Extension to Structured Methods of Design

"Structured Programming - improperly applied - is no better than traditional methods of program design" (Jensen 1981).

A cause of the GO TO controversy was the failure to correlate the symptom of the excessive use of GO TO's with poor designs. A mechanistic method of converting arbitrary programs to "structured programs" was developed - ie. the Ashcroft - Manna technique. (Yourdon 1975). However, this seldom increased logic visibility, and did nothing to improve the design itself.

The correct starting point for good programs, is a good design uninfluenced by the technology of the day. This would then be implemented in the most appropriate technology (Walker 1986).

As a result of this awareness, a new form of design was developed, which tended to produce structured programs naturally (Peters, Wirth 1974, Welsh and McKeag 1980, Jackson 1975).

2.3. Design Methodologies

The spectrum of design methodologies includes:-

- Structured Analysis
- Top-down Design, Coding and Testing
- Structured Design
- Structured Walkthroughs
- Modular Programming
- Various Logic Presentation Methods

2.4. Structured Analysis

Structured analysis is a set of techniques to define the functions of a system.

A detailed model of structured analysis is given in (Ross 1977, Yourdon 1979 p46, Wasserman 1980). Rigorous analysis is an important tool to

minimise error costs and to prepare for design. An error removed during the analysis phase can cost two orders of magnitude less than if removed during acceptance testing (Yourdon 1979 p 53).

The items are:-

- data flow diagrams
- data dictionaries
- written documents

2.4.1. Data Flow Diagrams

Data flow diagrams are a graphical representation of systems (Yourdon 1979 p 39, Shooman 1983, Ross 1977).

Related methodologies are Structured Analysis and Design Techniques (SADT) (Peters, Wasserman 1980, Ross 1985), Systematic Activity Modeling Method (SAMM) (Peters), which have formalisms for input, output, control and feedback.

The value of data flow diagrams lies in their ability to describe general systems in the form of abstract, logical models. By the use of hierarchical decomposition, the detail in each diagram can be limited. "Everything worth saying about anything worth saying something about must be expressed in six or fewer pieces" (Ross 1977). They can thus be used to define existing or proposed systems, manual or computer based.

A simple, but adequate, form is described by Gane (1980) and Gane and Sarson (1979).

Data flow diagrams define the data flows, data stores, data transformers (processes), and external entities.

Data flows are paths along which data is passed.

Data stores are repositories for data. These can be fixed (look-up tables), or can contain varying amounts of changeable data.

Processes transform data. They may consume or produce data.

External entities define processes which are outside the scope of the diagram. They may also produce or consume data.

No timing or synchronisation is indicated in these diagrams.

2.4.2. Data Dictionaries

The data *referenced in* data flows or data stores is defined in data dictionaries. The entries may take the form of top-down logical definitions of the data items, *detailed only* to the level required for the design. Attributes such as ranges and logical types may also be incorporated.

References include Peters, Gane (1980), Gane and Sarson (1979).

2.4.3. Written Documentation

The transformations performed by the processes need to be defined. The means of definitions include non-structured written *specifications*, structured English (Yourdon 1979 p 42), pseudo code or program design language specifications. These individual definitions form *"mini-specs"*, describing the *detailed* requirements and logic in terms of the defined data items (Yourdon 1979 p 42).

Structured English is a form of control-structured procedural or logical specifications. These *specifications* are intended to be concise, clear and complete definitions (Yourdon 1979 p 42, Gane 1980, Gane and Sarson1979).

The written specifications can be augmented by design trees (Bailey 1982, Peters), decision tables (Bailey 1982, Peters, Hurley 1983), cause-effect graphs (Myers 1976), Petri Nets (Agrawala 1979), State Diagrams (Walker 1986), graphs, and formulas.

The written documentation can also be used *to define* inter-process *dependencies* such as timing, synchronisation, priorities and data flow volumes.

2.5. Top-Down Design, Coding and Testing

2.5.1. Top-Down Design

Top-down design is an approach where the major functions are identified and defined before the *sub-functions*. This is then iterated with the sub-functions.

Alternative generation and selection is performed at each step.

Whilst a particular functional level is being considered, the sub-functions can be regarded as hidden or available functions on a hypothetical system (Shooman 1983). The final sub-functions are small solvable units.

Top-down design is known by a number of names including "programming by stepwise refinement" (Wirth 1971, Myers 1976, *"hierarchical design"* (*Constantine* in Yourdon 1975 p 36), and "levels of abstraction". A practical guide is given by Ledgard and Chmura (1978).

It may occur that this process results in a function which must be assumed as available, *that in fact is doubtful*. A check must then be performed. This check can take the form of detailed analysis, or a small test program. The intent of the check is to *increase confidence*, or to continue the search for a

suitable design alternative. The test program should not form part of the design.

The breakdown of a design for a particular level should not exceed a page (Yourdon 1975). This ensures that the major sub-functions only are defined.

This process is terminated when all the lowest level sub-functions have a trivial implementation. This implies that no design decisions are required during coding.

Interfaces and data must be defined at each level of function. The information includes the data passed to and from a function, as well as timing and data availability constraints. The data is defined in terms of the minimum detail applicable to that level. The data items are considered in logical terms, not implementation issues (such as word sizes or byte alignment).

2.5.2. Top Down Coding

This approach has *major function level code written* before that of the sub-functions. This can take place in parallel with design. Confidence in the final solution is increased, particularly if this code is run, perhaps with simplified sub-functions.

2.5.3. Top Down Testing

The classical approach to testing is to test each lowest level function using test driver routines. Once all the sub-functions for a function have been tested, the function itself is tested. This produces confidence in the lowest level functions, but results in the top level being tested last and least. Problems encountered at the top level may thus invalidate a major part of the sub-functions at a very late point in the project (Yourdon 1979, Myers 1976).

In opposition, in top-down testing the major functions are tested first, perhaps by simulating the lower level functions by test stubs. The result is that the problems encountered are *likely to affect a smaller part* of the design, and that the most crucial functions are tested most often. Where a sub-function is suspected to present problems, this may be checked in a similar manner as anticipated design problems, then reverting back to top-down testing.

2.6. Structured Design

Structured design is a design approach where the hierarchies of the program parts have emphasis. One such method is top-down design. Other forms of structured design result from decomposition using criteria of cohesion and coupling, transform-centred design (Yourdon, 1979 p 94), levels of abstraction (Myers 1976, Wasserman 1980) or information hiding (Parnas 1972, Peters).

Structured designs are typically illustrated by structure charts (Yourdon 1979 p 147, Peters), or HIPO (Peters, Wasserman 1980).

Structure charts are hierarchical diagrams, with rectangles demonstrating modules, connected by vectors *augmented by explicit data and control passing*. Selection and iteration are indicated, but not in detail. As a result of defining the data and control passing, module interfaces are defined.

2.7. HIPO

HIPO (Hierarchy, plus Input, Process, Output) is a documentation package extensively used and promoted by IBM (Yourdon, 1979 p 145; 153, Peters, Shooman 1983, Freeman 1980). The component parts are a hierarchy chart of the modules, and a module-by-module definition of inputs and outputs called a detailed HIPO diagram or functional HIPO diagram.

2.8. Warnier-Orr Method

The Warnier-Orr method is a graphical method of illustrating designs by bracketed sequence, selections, repetitions based on output data (Shooman 1983).

2.9. Jackson's Methodology

Jackson's methodology is a systematic constructive approach to programming. The input and output data of a process is defined using a graphical model of sequence, selection and iteration. A combined program structure is formed by the use of some defined techniques. The program structure is also given in terms of sequence, selection and iteration. Procedural fragments are defined for each operation and assigned to the structure. These fragments are combined into a pseudocode format called METACODE. The main reference is Jackson (1975). Others are (Peters, Wasserman 1980, Cameron 1983, Jackson 1976).

2.10. Structured Walkthroughs

The concept of walkthroughs is to check design or code in a systematic manner. The programmer himself, as well as people not necessarily directly concerned, form a team to check that the logic is correct, *specifications will be* satisfied, and that standards are met.

The use of structured walkthroughs results in the shift of responsibility of a program from an individual to a group. The group has a responsibility to assist the designer in finding errors or omissions, and indicating non-conformance to group norms and standards. The design is given high visibility. As a result the designer also tends to minimise non-conformance.

Structured walkthroughs are particularly effective at finding systematic errors. A designer may make an error, believing it to be correct. As a result, *he is unlikely to find the error himself.* Similarly *he may omit to* check a particular item as he is convinced of its correctness.

This psychological "set" or attitude is thus incorrect to find an error in that item. These problems are unlikely to be shared by the rest of the team, and the errors can more readily be found.

Weinberg (1971), in his "Psychology of Computer Programming" covers

structured walkthroughs and related methods such as ego-less programming.

Walkthroughs are often related to the phases of design used for a project. Thus specification, design, code and test walkthroughs, or system design, preliminary design, detail design and code walkthroughs may be defined.

Walkthrough techniques are well defined in the literature (Yourdon 1979 p 188, Shooman 1983, Freeman 1975, Fagan 1976, Zelkowitz, Yeh, Hamlet, Gannon and Basili1984).

The results of walkthroughs are the early reduction of errors (by a factor of about five Yourdon, 1979 p 193), improved quality, and improved design techniques (Fagan 1976, Boehm 1983).

This technique can readily be implemented in any organisation, provided that simple rules, such as impersonal checking, and timely checking, are followed.

A quantitive study of the effectiveness of structured walkthroughs was made by Myers (1978). Seven methods were used by 59 highly experienced professionals. The test was on a single PL/I program of 63 statements with 15 known errors. Code walkthroughs were as effective as any computer based error finding method. Greatest success was found by using two independent persons performing walkthroughs who then compared results. Significantly, even with this method a mean of only 8,3 of the 15 errors were found.

The term "structured" relates the design and code reviews to the strategy of the design and coding itself.

2.11. Modular Programming

This is one of the earliest approaches towards structured programming. Yourdon described modular programming in simple terms in 1972 (Yourdon 1972).

Modular programming consists of the division of programs into entities called modules. This rather inadequate definition is supplemented by various sub-division strategies. For example, the strategy can be based on size constraints (eg. 512 bytes or 4096 words) (Shooman 1983), complexity constraints (eg. one page of design or code), single function packaging (Ross, Goodenough, Irvine 1975) etc.

A typical definition is "The term 'module' is used to describe a largely self-contained section of the program which performs a specific function or sub-function" (Shooman 1983).

Useful guidelines in module definition are those of coupling and cohesion (Stevens, Myers and Constantine 1974).

Coupling is the degree of interdependence of the modules. The greater the independence of the modules, the greater the ease of understanding, designing, testing and maintaining the modules. Various levels of coupling are defined (Shooman 1983, Myers 1976).

Cohesion is the degree of interdependence of the components of a module. The

greater the degree of dependence, the greater the likelihood that the module performs a single function. Where the cohesion is low, the module is not a single function. Benefit would be derived by sub-dividing further (Shooman 1983, Myers 1976).

The principle of information hiding as a guideline is given in Parnas (1972).

The principle of designing for subset and superset implementation is covered by Parnas (1979).

The result of modular programming is a set of relatively small modules (5 - 50 statements). The strict specification of number of statements per module does not necessarily result in the best coupling and cohesion.

An analysis of modular programming is given in Shooman (1983), and its use on complex systems by Parnas et al (1985).

2.12. Design Logic Presentations

Once components (top-down design), modules (modular programming), processes (data flow diagrams), or functions (structured design) are isolated, their design details must be put to paper. The procedural definitions is addressed directly in the structured design method.

A concise exposition of a number of methods used is given by Peters (1980), and in more detail in his book (Peters).

In many cases the design logic is not directly documented. The logic is defined only in the implementation directed code (Privitera 1982).

Techniques to define these details are:-

- flowcharts
- pseudocode
- Nassi-Shneiderman charts
- program design languages

Flowcharts

Flowcharts are the traditional manner of indicating logic. They are used for software documentation as well as a number of other uses. These include hardware logic, process descriptions, test and assembly procedures. As a result, they are not only widely used, but also readily understood by non-computer people.

Unfortunately, flowcharts are less suited for design than for documentation. Flowcharts are often illogical and complex as a result of arbitrary control flow.

Flowcharts have lost popularity, in direct relation to the growth in structured techniques. They do still have a role in illustrating procedures, particularly to non-computer people (Shooman 1983, Peters, Wasserman 1980).

Pseudocode

Pseudocode is a control structured subset of English. It has a relaxed, but similar, syntax to block structured programming languages. The degree of syntactic and semantic exactness is matched to the point being demonstrated. Most algorithms are illustrated in this manner (Shooman 1983, Zalkovitz, Shaw, Gannon 1979, Peters, Jensen 1981).

No standards exist for pseudocode, but most are based on sequence, selection and iteration constructs, with indentation. In fact, the pseudocode format usually bears a close relationship to the user's favourite programming language.

Whilst lack of standardisation between groups may not be that serious, unfortunately standards may differ within one project or program.

Nassi-Shneiderman Charts

These charts are a graphical method of designing and documenting programs. The key difference between Nassi-Shneiderman (or Chapin) flowcharts and other types, is the constraint to only structured constructs and abstraction.

These flowcharts are also a powerful synthesis tool by encouraging systematic decomposition.

These charts have become well established in many organisations (Siemens, UNISA) and are well documented (Yourdon, 1979 p 155, Peters, Freeman 1980, Yoder and Schrag 1978, Chapin 1974).

Machine support does exist (eg. AIDS package), but is not very effective because of formatting problems.

Program Design Languages

Program design languages are similar to pseudocode. They are also relaxed syntax languages. However, they have two components

- a fixed, formal outer (or control) syntax
- a free-form, uncontrolled inner syntax.

As a result, program design languages can be machine supported.

Program design languages may also support data and abstraction. Furthermore the machine supported forms can provide formatting, indentation and a variety of referencing forms.

Program design languages are considered in more detail in a further section.

3. SOFTWARE ENGINEERING

3.1. Definition of Software Engineering

"The term software engineering is used today to describe a loosely coupled collection of practices, techniques, and methods" (Peters).

"Software engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation" (Boehm 1981).

In the late 60's, the term "engineering" was increasingly associated with the practice of software. The term "software engineering" was at first a provocative phrase, to contrast the formal science to the pragmatics of engineering.

Software has begun to have a significant social impact. By 1985, 40% of American labour relied on computers and software to do their work, without being directly involved in software or computers (Boehm 1981).

This was, to a large extent, the result of increasing awareness of the need to change direction from a research or laboratory science to the systematic success of real projects. This awareness came as a result of major failures. These are examined in some detail in a further section. An analysis of failures is essential to find the causes, and how to prevent or overcome them. The tools, methodologies and practices must be chosen and designed explicitly to avoid them.

Real projects have considerable constraints in terms of cost, schedule and technical performance (or system quality). These constraints lead to a spectrum of requirements that must be satisfied. Optimisation is not normally required.

Real projects also require management visibility. Evaluation of a projects progress in terms of cost, schedule and performance is necessary to reduce risk.

"It comes as a distinct shock to the uninitiated that, for an activity that accounts for the expenditure of several billion dollars a year in the United States alone, the management of computer programming is still something of a black art. For all purposes there are no generally accepted, generally available, or generally applicable guidelines or techniques on which a manager can rely (other than the ad hoc experience of technical experts) in making cost or schedule estimates, in weighing the investment of increments of time and money against hoped-for improvements in the performance or quality of the computer program and product, or in assessing cost-to-value after the fact" (Weinwurm 1970).

These aspects were not traditionally seen as a part of software development. All significant projects now have at least some management methods associated with them.

The imposition of management constraints on the practitioner have often been met with opposition. They are deemed to interfere with creativity, and not to

be relevant to software. However, according to Metzger (1973), the management of software development is closer to the management of other disciplines than software staff would accept, yet further apart than management would accept.

Boehm (1983) has identified seven principles considered essential for long-term success in software engineering:-

- Manage using a phased life-cycle plan
- Perform continuous validation
- Maintain disciplined product control
- Use modern programming practices
- Maintain clear accountability for results
- Use better and fewer people
- Maintain a commitment to improve the process

Software engineering extends from the time that a problem or need is discerned, to the time that the solution is discarded. A number of phases within the development of software are defined. This enables discussion of the role that design tools, and program design languages in particular, play during each phase, and which needs they fulfil.

3.2. Management of Software Projects

To control progress, and report on its achievement, measurements must be performed. A typical method is the use of Work Breakdown Structures, Statements of Work and Project Planning and Reporting Methods (Boehm 1981, Shooman 1983).

The work breakdown structure is the hierarchical decomposition of the project into identifiable products or phases. It can be related directly to a design tree (Peters). The statement of work defines the contents and extent of each of the products or phases. Coupled with each is a definition of pre-requisites, resources and a budgeted cost.

Project planning and control methods can now be applied - for example CPM or PERT, and the reporting of actual progress and forecast progress against these plans and original estimates.

Up to 50% of unsuccessful projects have failed because of poor planning (Metzger 1973, Boehm 1981).

These methods provide insight into cost and schedule performance, and can be coupled to technical performance. This may result in further effort being directed to lagging items, techniques to improve design and implementation achievements (Weinberg and Shulman 1974), and may also be used to terminate sub-optimisation. Project management is covered extensively by Metzger (1973).

3.2.1. Lifecycle Models

The evolution of a software project follows a path from need to an accepted solution. Many models exist of the phases within this path, and what each phase entails (Boehm 1981, Shooman 1983, Boehm 1983). No concrete boundaries exist, and phases may overlap. As insight is achieved, previous phases may need to be amended. As a result, the phases are not absolute, do overlap, and may be iterated.

Within each phase a number of alternatives may be generated, evaluated, and a single choice made.

Typical stages in the lifecycle from need to transfer of a successful solution to the customer are:-

- Specification
- Conceptual design
- Detail design
- Implementation
- Testing

These phases are now examined in greater depth:-

3.2.2. Specification

In this stage a concerted effort is made to understand the user requirement, and to move from a broad based set of requirements to a more concrete detailed specification.

3.2.3. Conceptual Design

In this stage initial steps are taken towards the formulation of a concrete solution. The emphasis is on what functions the system must provide. Checks must be made to determine whether these functions are achievable. Analysis, formal and informal, may be required to satisfy these question.

The outcome of this stage is a set of proposals which are coherent and appear to satisfy all the requirements.

3.2.4. Detail Design

Each function defined during Conceptual Design is treated and goes through the stages of Specification, and Conceptual Design iteratively until no design decisions remain.

As a result of investigating these functions, further problems may become evident, and increased knowledge of the functions of the system is gained. As a result, the prior stages' output may need to be revised.

The output of this phase is a single coherent description of one solution, in logical terms. The detail is such that no further design decisions exist.

3.2.5. Implementation

In this stage the logical design, with some functions optimised, must be realised on the system available. Integration of the design concept with the peripherals and operating systems is performed.

The adjusted logical design must now be coded into the language, or languages, used on the system.

The code is compiled and linked to form a package with no compilation errors.

The output of this stage is a complete package of software.

3.2.6. Testing

"Testing is the process of executing a program with the intention of finding errors" (Myers 1976).

In this stage, the software package is verified operationally, particularly against the system's requirements.

Testing is covered in detail in (Myers 1976, Shooman 1983).

The errors found are resolved, and the outputs of each previous stage updated.

Testing should also be recognised during design. Some consideration of how to test, and what to test, can influence design beneficially.

Design is regarded as the activity covering the phases of conceptual design and detail design. Some overlap exists with other phases. Specification errors and omissions that are detected, are rectified in design.

Similarly, certain implementation constraints must be recognised during design.

3.3. Software Design Failures

The failures of large projects can be spectacular. In the final account they tend to exceed budget, be late, and not fulfil user requirements (Brooks 1975, Wesserman 1980). Furthermore, these failures are often only apparent very late in the project.

These are the two gross symptoms:-

- budget and schedule overruns
- failure to perform to specification

The budget and schedule overruns are as a result of over-optimism, and lack of planning and reporting to plans. The failure to perform to specifications is the result of incomplete specifications, lack of commitment to specifications,

and human error.

"... software is too expensive and software is unreliable. Most computer professionals recognise the former problem as largely a symptom of the latter (Myers 1976). An example of the fatalistic view: "Hence, plan to throw one [design] away; you will, anyhow" (Brooks 1976).

Methods evolved to minimise errors, reduce cost of error correction *and make* errors visible.

3.4. Human Constraints

"The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the *programming task in full humility* ..." (Dijkstra 1972).

A major input into the *methods to minimise failure was* the increasing awareness of the limited capabilities of humans.

Studies were made of *human capabilities in the area* of design. Clear areas of concern were found. A programmer's capability is quickly exceeded by the complexity of the systems he designs unless specific steps are taken (Walker 1966, *Miller* 1956). Also his capabilities are generally far short of his own expectations.

Limits in capability include systematic optimism, uncontrolled growth of complexity beyond the programmer's capability, and the mindset of a designer in missing his own logic errors.

"Students and professionals alike tend to be overly optimistic about their ability to write programs or to make programs work according to pre-established design goals" (Ledgard and Chmura 1978). This is further discussed in Brooks (1975).

Further references to the study of human capabilities and constraints are covered in "The Psychology of Computer Programming" by *Weinberg* (1971), Dijkstra (1972), Bailie (1982), and design issues are examined by Freeman (1984).

3.5. Cost Versus Phase in which Errors are Removed

A high correlation exists between cost of *error correction and* the stage of development in which it is found. Several orders of magnitude of cost separate error correction during specification and after delivery to site. (A D'Agapeyeff in Yourdon *1973 p 243, Boehm* 1981, Shooman 1983, Myers 1976, Boehm 1983).

A consideration of failure modes in each phase is relevant.

During the specification phase, a semantic gap is common between the customer *and designer.*

The customer typically shows a bias towards his application area knowledge, *allows a maximum scope* of acceptable solutions to maximise the range of

offered solutions, yet excessive restrictions in solutions to remain within his area of comfort.

Finally the customer generally provides insufficient information, and is ill-equipped to monitor and control development.

The designers in turn have insufficient awareness of the application area, and provide insufficient insight into the evolving design from the customer's viewpoint and in terms of his knowledge.

During the conceptual design, insufficient communication takes place between customer and designer. At this stage the wrong system may already be chosen. No documentation is generally produced and accepted during this phase.

During the detail design phase, ad-hoc methods are generally used, whilst often the "design" is done while coding. Clear, logical documentation with an appropriate notation would make these errors visible, as well as indicating concrete progress.

At this phase the designer generally becomes lost in a mire of details (Walker 1986), and loses his perspective of the importance of his decisions.

During the coding phase, poor discipline results in monolithic masses of code, with little support documentation.

In fact, this phase is the least difficult, provided that all design decisions have been made. The gap between design and code should be the same as between the code itself and code in a different programming language.

The phase of testing is generally the one where failure becomes apparent. This phase then often stretches in duration, in many cases to over half of the total project duration. Errors as a result of insufficient discipline in all the previous phases are corrected, often resulting in a cascade of induced errors. Studies have shown that changes of five to ten statements have the best probability of being correct - and that is only 50% for each attempt (Yourdon 1979).

The testing is often undirected, with poor error detecting. This is the result of not considering what the tests, and results of the tests should be, from the earliest phases of the project. Furthermore, the conceptual design and detailed designs can be desk checked against requirements from the earliest phase.

If testing is performed throughout the project, at this phase the bulk of the testing is to confirm that the code directly represents the checked detail design. Certain expected and unexpected errors may still surface during a sequence of pre-planned tests verifying operation to specifications.

It must be emphasised that an error in any of the phases must be correlated with the other phases' output to ensure total correspondance.

This analysis indicates that reducing errors in the design phase is cost effective. In particular, error minimization during the design phases ensures minimum errors during implementation and testing. Consideration of testing

methods and detailed procedures is also useful.

Although *specification errors and omissions* occur before the design phases, effort should be given to analysing the specifications during design. Clarity and completeness of the design presentation can highlight specification problems.

3.6. Techniques to Reduce Errors

It became apparent that the use of clear, logical design documentation with technical reviews, or structured walkthroughs, was a major tool to weed out errors. The retrospective approach of trying to remove errors once *in the* final code is not simple. Dijkstra summed up that situation by "Testing shows *the presence, not* the absence, of bugs" (Yourdon 1975 p 244).

Other techniques developed to remove errors were:-

- close relationship to customer
- use of structured techniques
- logic emphasis
- complexity management
- modularisation
- egoless team approach

3.6.1. *Customer Relationship*

A technique of minimising certain errors, particularly those of requirements and functions, is to *form a combined team of customers* and designers. This would increase confidence and reduce risk, and tends to keep both sides honest. A greater knowledge transfer is encouraged - application knowledge to the designers, and *implementation knowledge to the customer.* The transfer of the system to the customer is also eased (Myers 1976).

All customer - designer problems are not however *solved by this approach.*

3.6.2. Use of Structured Techniques

An analysis of factors affecting productivity provides insight into the *benefits that* can be expected from the use of structured techniques. This is summarised in Yourdon (1979 p 34):

Range of Projects in Survey

	Low	High	Median
Size of System (Lines of Code)	900	712 000	21 000
Manpower (Months)	3	11 760	60
Duration of Project (Months)	1	68	10

Table 1

Effects of Structured Techniques on Productivity

	Productivity (Lines of Code per Month)		Product-ivity Ratio
Use of Techniques	Not Used	Used	
Structured Code	169	301	1,8
Top-down Development	198	321	1,6
Code Reviews (Walkthroughs)	220	339	1,5

Table 2

Error rates, where best use of structured techniques is made, can be reduced to an average of one to five errors per 10 000 statements (Yourdon 1979 p 35).

3.6.3. Logic Emphasis

Logic emphasis is an approach to provide an abstract solution.

In order to produce a successful software system, the design must be successful. A successful design is most easily assured by making the logic visible and clear. A design can then be checked against the specification. A further separate phase of optimisation may then be recognised and included, once the logic is seen to be correct, and that the direct algorithm does not meet speed or space constraints.

3.6.4. Complexity Management

Complexity management consists of a concious effort to reduce a solution's complexity whilst recognising the lower bound of the problem's complexity (Myers 1976).

A large project is more subject to high complexity - the effort is a higher-than-unity power of deliverable source lines (Boehm 1981, Dijkstra 1972).

"The design and development of information processing tasks requires a high level of self-discipline simply because in the nature of things it is a human

characteristic to be woolly, unstructured and illogical in our thinking " (Walker 1986).

Methods were developed to reduce complexity. These included the use of minimum control constructs, *modularisation, abstraction and modelling.*

Complexity metrics have been examined in an empirical manner (Halstead 1977, McCabe 1976, Curtis 1980).

Systematic efforts to reduce the complexity of a design item were applied. This was achieved by reducing coupling, increasing cohesion, use of symbolic logic, definite rules to minimise module sizes, and to separate logic from implementation (Walker 1986).

3.6.5. Modularisation

Modularisation or the use of small modules, and the disciplined use of standards can minimise problems in this phase. Code inspections and desk checking can enforce these standards and find many errors.

3.6.6. Egoless Team Approach

Egoless teams (Weinberg 1971, Myers 1976) are an organisational structure and approach which removes the individual's *sensitivity regarding his "own"* programs and designs, subjecting them to the evaluation and constructive criticism of his peers. The group ownership of the effort is intended to ensure that the best talents of all concerned are embodied in the resulting designs and code.

4. DESIGN TOOLS

4.1. Definition of Design Tools

Design tools are aids to the process of design. They are complementary to methodologies and practices. Design tools assist the designer, either by reducing effort, or by directing effort.

Key aspects to design tools are:-

- mechanised support
- user friendliness
- compatibility to methodologies and practices of design
- compatibility to implementation methods

Other attributes of design tools are machine independence, operating system independence, target system independence, ease of use, learning overhead, graphical presentation.

A design tool can be directed to design documentation, design synthesis, design view presentation, and design relation illustration.

Some examples of design tools and design representation schemes are given in Shooman (1983).

4.2. Typical Design Tools

The design phase considered here is between the specification phase and the implementation (or coding) phase.

Some tools are:-

- Pretty Printers
- Data Flow Diagrammers
- Nassi-Shneiderman Packages
- Program Design Languages

4.3. Pretty Printers

Pretty printers are formatters which take unformatted source and produce indented, paginated output.

Known pretty printers are all target language oriented (Rubin 1983) though they could be designed to be more general purpose. As such they are not currently design tools.

4.4. Data Flow Diagrammers

Various proprietary documentation tools exist for data flow diagrams. Certain specialised methodologies such as SADT (1985) have machine supported tools under development. They are not generally available.

4.5. Nassi-Shneiderman Packages

One package is known to produce Nassi-Shneiderman (or Chapin) flowcharts known as AIDS. This package uses a control structure subset of Intel's PL/M language, possibly of a full PL/M program. According to the constructs, the associated conditionals and statements are formatted into the required sequence, selection and iteration blocks. Restricted support of GO-TO's is also provided.

This package is oriented towards the reformatting of source code. It is highly machine and language dependent.

4.6. Program Design Languages

A program design (or documentation) language is a syntactically formal language. Its form is similar to pseudocode, or Structured or Tight English. Two parts exist to the syntax - an outer, control structure oriented syntax, and an inner, free format statement syntax.

The outer syntax supports sequence, selection and iteration. These constructs also control the indentation of the output. In this aspect, a high correlation exists with pretty printers.

The outer syntax is augmented by formatting controls, and data and segment definition controls.

The inner syntax is generally uninterpreted, but is formatted as a result of the surrounding outer syntax.

The algorithmic logic of a design can thus be defined using DO - END, IF - THEN - (ELSEIF -) ELSE - ENDIF, DO - WHILE, and DO - UNTIL type constructs, with English statements taking the position of conditionals and statements. The English statements can then be procedural or functional definitions.

The use of the free-format inner syntax provides the tolerant richness that makes a program design language such a powerful, general purpose tool.

The outer syntax provides consistency and standardisation which improve clarity and understandability.

Abstraction can be provided explicitly (using CALL statements) or implicitly (by naming all defined program segments). The package then relates these statements wherever they occur.

In a similar manner, data elements and structures can be defined explicitly (in data definitions), or implicitly (for example by control characters in their names).

5. PROGRAM DESIGN LANGUAGES

5.1. Definition

A program design language is a two-level language capable of expressing design logic. In form it is similar to pseudocode or a programming language. One level - corresponding to control structure such as those for sequence, selection or iteration - is formal, like a programming language. Extension to include concurrency is possible, but is not covered here (Hoare 1978). The other level is informal, and allows the free expression of design, without restriction to valid statements and operators as in a programming language (Zelkowitz, Shaw and Gannon 1979, Peters, Caine and Gordon 1975, Wasserman 1980, Freeman 1980, Wasserman 1982, Burstin, Porsher, Malmon, and Rothard 1983, Ramanathan and Blattner 1979, Sutton and Basili 1981).

The formal outer syntax allows the design description to be automatically processed. The processing formats the design according to standards for indentation and pagination.

No enforcable standards exist for pseudocode (or Structured English or Tight English). However, the definition of a program design language, and its formatter, do form a standard.

5.2. Relationship to Lifecycle Phases

A program design language is used to express logic. This function is useful in various phases:-

- specification of detailed requirements
- conceptual definition of tasks, data entities
- detailed design of tasks and data entities
- implementation of code by providing all design decisions in a concrete form
- testing of software by providing a logical reference of operations
- maintenance by providing design insight

5.3. Use of Program Design Languages

A program design language can be used as soon as any procedural or functional specifications are tied to a process, or as soon as data elements or structures are identified. The logic can then be expanded in a top-down, or stepwise refinement, manner.

Because of the tolerance of the language, properly formatted design documentation is possible once a few statements have been defined.

This documentation can then be expanded, updated and revised as knowledge of the design increases, and the details are defined.

The design is thus available from the beginning for design reviews and structured walkthroughs.

The aspects of design that are supported are the logic, data structure, and input - output specification.

5.4. Form of Program Design Languages

5.4.1. Outer Syntax Level

One level is a semantically exact outer syntax, which defines the structure of the language.

This outer syntax typically defines logic entities, their control constructs and abstraction (eg. procedure CALL's) . The outer syntax can also define data entities, their elements, and subservient data entities (eg. arrays, structures, files, manifest constants (equates or literals)).

The outer syntax is indicated by use of reserved words ("IF", "THEN", "ELSE", "ENDIF" etc), or by control statements (used, for example, to set the date of a print-out).

Coupled to the outer syntax, is the indentation and pagination of the output. The relationship between named entities and their invocations results in the calling hierarchies and cross references that can be provided.

5.4.2. Inner Syntax Level

The inner syntax consists of strings that are interpreted in terms of the surrounding outer syntax. The main constraint on the inner syntax is that it does not clash with the outer syntax (reserved words etc. must not be used). Otherwise any free-format text may be used.

The inner syntax, in conjunction with the outer syntax, is used to define the logic. As the inner syntax is free-format, it can express any logic directly or indirectly by invocation of a further defined entity.

5.5. Typical Program Design Languages

Four languages are examined in some detail:-

- Van Leer's
- Caine and Gordon's
- SuperPDL
- Walker's

5.6. Van Leer's Program Design Language

5.6.1. History

This language was developed by McDonnell Douglas Automation Company during the early 70's (Van Leer 1976). The company had a number of analysts who prepared specifications for programmers. The varying approaches, degrees of completeness and correctness posed problems for the programmers. In order to benefit from the structured techniques and to standardise approach, a policy was formed. This policy specified the use of structured walkthroughs, program design language, top-down programming and structured programming. These successful techniques were supplemented by the use of HIPO.

The language was also used for training purposes.

5.6.2. Program Syntax

The syntax was defined to be compatible to PL/I and Cobol. Single entry and *exit routines of no more than one page are defined.*

The standard constructs are supported. They have the form of:-

```
* IF [condition]
        THEN [statements]
        ELSE [statements]
   ENDIF

* DO WHILE [condition]
        [statements]
   ENDDO

* PERFORM UNTIL [condition]
        [statements]
   ENDLOOP
```

Multiple conditions or statements are aligned one below the other. The THEN or ELSE part may be omitted.

Sequence is implied by concatenation of constructs.

The CASE construct is also implemented:-

```
* CASE [variable] OF
        [Value one]: [statements]
        [Value two]: [statements]
        .
        .
        .
        [Value n]: [statements]
   ENDCASE
```

Multiple values may be given on one line, delimited by colons.

The constructs may be nested. The current construct reserved word is directly followed on the same line by the reserved word of the nested construct. For example:-

```
* IF [condition] THEN DO WHILE [condition] [statements]
```

ENDDO ELSE [statements] ENDIF

Abstraction is implied by creating a routine corresponding to the invocation.

5.6.3. Data Syntax

No data capabilities are provided.

5.6.4. Formatting

A unit of indentation is applied consistantly (eg. 3 spaces).

5.6.5. Referencing

No facilities are provided

5.6.6. Mode of Operation

No mode of operation (batch or interactive) is defined.

5.6.7. Availability

No computer support is defined in the paper.

5.6.8. Special Features

No additional features are defined.

5.6.9. Formality

Beyond the constructs, no additional formalism is defined.

5.6.10. Purpose

The program design language is particularly targeted towards detail design using structured techniques, for implementation in PL/I or Cobol.

5.6.11. Results

No specific evaluation is presented for the program design language in isolation. However its symbiosis with the other techniques, and their combined success, resulted in a policy decision to use these methods.

5.6.12. General

This is the classical paper on program design languages. The discussion of their applicability, goals and purposes is worth reading.

5.7. *Caine and Gordon's Program Design Language*

5.7.1. History

This language was developed by Caine, Farber and Gordon during 1973 (PDL Program Design Language Reference Guide 1977). The language was to

supplement programming techniques by support during the design phase. The techniques used were structured programming, top down design and implementation, centralised libraries and egoless teams.

5.7.2. Program Syntax

The syntax is oriented towards PL/I or Algol-like languages.

The standard constructs supported are:-

```
* IF [condition]
        [statements]
  ELSE
        [statements]
  ENDIF

* DO [condition]
        [statements]
  ENDDO
```

The ELSE is optional.

The [condition] of the DO construct may have a prefix of "while" or "until". Use may also be made of a "case" prefix. The construct may also have UNDO (passing control to after the matched ENDDO), or CYCLE (passing control back to the [condition]).

The syntax is augmented by the use of explicit labels taking the form of:-

* [label]: The colon may be followed by a normal line.

The syntax has two forms:-

- the entry format

- the listing format

The entry format consists of the reserved word, followed by appropriate free-format text. The reserved words include IF, ELSE, ENDIF, DO, UNDO, CYCLE and ENDDO.

The listing format consists of underlined reserved words and text, formatted according to its rules.

The constructs may be nested, but only one reserved word is given per line.

Abstraction is directly supported. Routines (or flow segments) are explicitly initiated by the use of a prefix (%S). The line of text then defines the routine's name. The routine is delimited by the next control statement (eg. the next %S). Sequence is thus supported.

5.7.3. Data Syntax

Data can be defined explicitly or implicitly. A data definition segment can

be defined similarly to the flow segment, by use of a %D prefix. The subsequent word is regarded as a data item name. This is specially noted.

Data can also be implicitly defined within flow segments by including at least one underscore in the name. The name will then also be noted.

A low-level support for external definition is also provided.

5.7.4. Additional Syntax

Support is also provided for text blocks. These are initiated by a %T prefix. No formatting is performed.

Many options exist. Headings, dates, group identifiers may be defined. Line and page lengths may be set. Prefixes and special characters may be altered.

5.7.5. Formatting

The listing consists of front page, table of contents, group identifiers, segment pages, calling hierarchy tree listing, and flow segment and data segment indexes.

Only the segment pages are directly entered by the user.

The flow segments are automatically indented, reserved words underlined, and invocations referenced.

5.7.6. Referencing

The table of contents is automatically generated.

The calling hierarchy tree listing gives an indented listing of flow segments. Recursion is recognised, and a short tree option does not repeat common subtrees.

The flow and data cross references provide references to definition and usage page and line.

Lines and pages are automatically numbered.

5.7.7. Mode of Operation

This language is supported by a batch oriented processor. A free-format, concise source is edited. This is then processed ("compiled") by the processor. A listing is then produced, with error indications.

5.7.8. Availability

The processor is commercially available, and is supported by various processors including IBM 370, DEC PDP's and VAX.

5.7.9. Special Features

This language is very complete. Many options are available.

5.7.10. Formality

The listings are easily interpreted as a result of minimal, and self explanitory, constructs. The use of a special character for data words requires some acclimatisation. The perceived formality is low.

The input format is quite formal. Free-format is supported. Concise controls are used.

5.7.11. Purpose

The language is oriented to global and detail design for routines as well as data.

5.7.12. Results

The authors indicate a 50% improvement in software productivity with use of the language. Some additional statistics are provided. Their results are similar to the results I have experienced with approximately 10 man-years of usage in an imbedded system development environment. The comment by the authors is however relevant:-

"PDL is not a "panacea" and it is certainly possible to produce bad designs using it. However, we have found that our designers and programmers quickly learn to use PDL effectively. Its emphasis on designing for people provides a high degree of confidence in the correctness of the design. In our experience, it is almost impossible to "wave your hands" in PDL. If a designer doesn't really yet see how to solve a particular problem, he can't just gloss over it without the resulting design gap being readily apparent to a reader of the design. This, plus the basic readability of a PDL design, means the clients, management, and team members can both understand the proposed solution and gauge its degree of completeness.

We have also found the PDL works equally well for large and small projects. Because it is so easy to use, persons starting to work on even a "quick and dirty" utility will first sketch out a solution in PDL. In the past, such programs were usually written with little or no design preceding the actual coding".

5.8. SuperPDL

5.8.1. History

This language was developed by Advanced Technology Ltd as a result of in-house needs. A language was needed that exceeded the capabilities of that by Caine and Gordon (1975). The 350 man-year project on hand necessitated an *interactive system providing concurrent facilities for many users.* Inter-module interface support was considered highly important.

5.8.2. Program Syntax

Sequence, selection, iteration and abstraction constructs are provided:

- Sequence is implied by concatenation of constructs

- Selection is supported by IF-THEN-ELSE and SELECT

```
    * IF [condition]
         THEN
              [statements]
         ELSE
              [statements]
          ENDIF

    * SELECT [selector]
         ON [valueone]
              [statements]
         ON [valuetwo]
              [statements]
              .
              .
              .
         ON [valuen]
              [statements]
         OTHERWISE
              [statements]
          ENDSELECT
```

- Iteration is provided by FOR, WHILE, UNTIL and EXIT

```
    * WHILE [condition]
        [statements]
         ENDWHILE
```

- Abstraction invocation is by explicit CALL and related named module:-

```
    * CALL [module name]
         INPUT
              [parameter list]
         OUTPUT
              [parameter list]
          ENDCALL
```

5.8.3. Data Syntax

Data and abstract types are supported:-

```
    * DEFINE
         TYPE [type name]
              [data items][short descriptions]
         TYPE [type name]
              [data items][short descriptions]
              .
              .
              .
          ENDDEFINE

    * DATA-TYPE NAME: [name]
         TITLE: [extended name]
         OPERATORS:
```

[operator names][operator descriptions]
FUNCTIONS:
[function names][function descriptions]
END-DATA-TYPE

5.8.4. Formatting

Several reports are produced including component (external) definitions, module (routine) descriptions, structure chart (calling hierarchy tree listing), cross references, full system description (ordered routine and data listing), and lists of routines with various attributes.

5.8.5. Referencing

The referencing capabilities are equivalent to those provided by Caine and Gordon (1975).

5.8.6. Mode of Operation

The processor is an interactive system using a syntax-directed editor prompting with construct templates. Where a CALL template is defined to correspond to an existing routine, parameter matching prompting is provided.

5.8.7. Availability

This processor is commercially available.

5.8.8. Special Features

The processor is suitable for a concurrent multi-user environment on single large programs.

5.8.9. Results

The results achieved by the designers are:-

- low training requirement
- immediate acceptance
- improved productivity
- integration time reduction
- reduced documentation effort

5.9. Walker's Program Definition Language

5.9.1. History

This language was developed by A J Walker as "The need was felt for a logic verification aid which would require minimal text entry for network description and simulation execution purposes" (Walker 1964, 1986). This language has emphasis on modeling and simulation, and is syntactically and

semantically more concrete than the other languages considered. The language is used as a basis for a course in system design covering a scope from hardware logic, microprogrammed logic, assembly language to high level language. Concepts such as data flow, parallel processing and process-resource are are illustrated. Several research projects are being performed and promoted in this area, including an interactive program definition editor (Bassanino and Walker 1986). Future projects are directed towards direct generation of high-level language programs.

5.9.2. Program Syntax

The standard constructs are supported:-

```
    * IF [condition]
        THEN:
            [statements]
        ELSE:
            [statements]
        ENDIF:

    * REPEAT:
        [statements]
        UNTIL:      [condition]

    * WHILE [condition]
        DO:
            [statements]
    ENDWHILE:
```

The "ELSE:" part may be omitted.

The "CASE" construct is supported:

```
    * CASE [variable] of:
        [valueone]:[statements]
        [valuetwo]:[statements]
        .
        .
        .
        [valuen]:[statements]
        ELSE: [statements]
        ENDCASE:
```

The "ELSE": part is optional.

Abstraction is supported by invocation and the related procedure definition.

```
    * CALL [procedure name] ( [input parameter list] ) [output parameter
    list]
```

A complete syntax is defined, including assignment, input/output, arithmetic and logic operators, sequential and parallel execution, and comments.

5.9.3. Data Syntax

The support for data structures is extensive. Characteristics defined are:-

- function : constants or variables

- type : elementary (boolean, integer, real and character) or composite (defined in terms of elementary or other composite types)

- structure : "Single" (an item of a particular type) or "Array " (multiple items of one particular type)

- scope : accessibility ("local" to procedure, "external" to a procedure, and "global" to procedures within a partition), and lifetime ("permanent" where the data structure values are retained between invocations).

- name : the identifier

These characteristics are all requisites for data structures. In addition, a data description segment is required for programs or procedures. A predefined order of characteristics exists.

5.9.4. Additional Syntax

Support is given for tasks, programs, procedures, processes, resources, etc.

The syntax is complete enough to map into high level languages, and to model systems of various kinds.

5.9.5. Formatting

Indentation is used. Because of the degree of data definition, the length of procedures is not restricted to a page.

5.9.6. Referencing

No referencing forms are defined. These would be implementation dependent.

5.9.7. Mode of Operation

The language defined is not bound to a particular mode of operation. Manual formatting is possible, and an interactive processor exists (Bassanino and Walker 1986).

5.9.8. Availability

A version exists for DG Eclipse, and one is under construction for a personal computer.

5.9.9. Formality

This language is the most formal considered. Its may be regarded as a potentially executable language. Automatic generation of an equivalent high level language program is under consideration.

5.9.10. Purpose

The language is used for training, modeling and simulation, and to provide verifiable, complete designs in a technology-independent form.

5.9.11. Results

The language has been successful within its application area. As a result of experience, the language is likely to grow.

5.9.12. Modes of Operation

Two basic operational modes are applicable:-

- batch operation

- interactive operation

Batch operation consists of the "compilation" of a source containing the statements as well as certain controls (for formatting etc.). The result of the "compilation" is a printable file in paginated and indented form, with error statements as required.

Interactive operation is based on syntax-directed entry (Bassanino and Walker 1986). A syntax-directed editor knows the syntactic rules of the language. To enter a design, the main construct is selected. As a result, that construct appears on the screen, with placeholders for conditionals and statements. Syntactically invalid entries can thus not be made. Other commands exist to enter the conditionals and statements, as well as to update and modify existing information. At every stage, the constructs are indented to reflect their structure.

5.9.13. Batch Oriented Systems

A batch oriented system is particularly amenable to rigorous configuration management, by controlling baselines. A particular baseline is defined, and printed. All changes are then indicated with reference to this version, which may be distributed to a number of users. At a particular point in time, these changes can be consolidated into a new baseline.

A batch oriented system, by its nature, encourages disciplined changes - minimising the "design at terminal" syndrome.

More sophisticated additional output is also possible. Line and page numbering, line-by-line cross referencing, calling hierarchy tree indexes, and segment and data cross references can more readily be produced.

Finally, a batch oriented system is easier to construct. System utilities

such as screen editors can be used to prepare the source, reducing the effort in designing this critical man-machine interface. *The additional effort* can be applied to the indexing and cross-referencing.

Attention given to compile efficiency would reduce the turn-around time required.

5.9.14. Interactive Systems

An interactive system allows greater user friendliness, particularly for the novice or infrequent user. Syntax errors cannot be introduced. A menu system can prompt the choice of next construct to be used. The use of sophisticated windowing techniques would allow the direct use of abstraction. A concurrent semantic analyser could relate attributes and allow viewing of all symbolic references. Certain error types relating to semantics could also be determined - uninitialized variables, unused manifest constants.

The interactive systems lend themselves to preparing quick solutions, and to presenting changes. This may lead to radically shifting baselines and lack of change control. Also the ease of changes may produce unstable designs with high residual errors. "Design at terminal" may be encouraged.

The role of organisational and *self discipline must not be underestimated.*

An interactive PDL system is described in Bassanino and Walker (1986).

5.10. The Value of Program Design Languages

Program design languages are effectively the only design tools that have achieved widespread use and success (63% of companies in a survey) (Zelkowitz, Yeh, Hamlet, Gannon and Basili 1984). In fact, the use of the programming design language is the first choice for a design tool by the authors.

The use of program design languages as design tools may provide benefits in the areas of:-

- early documentation
- visibility
- communication
- early error removal
- checking procedures
- front-end loading
- productivity
- reusability

These areas are discussed in more detail.

5.11. Early Documentation

Documentation is generally regarded as low priority, uninteresting and rewardless work by designers (Metzger 1973). Management is generally concerned with the high cost of good documentation. As milestones are missed, the temptation to reduce, ignore or defer documentation is often irresistable.

Unfortunately, inadequate documentation in terms of completeness, quality and timeliness increases risk of a project enormously. Many projects have been re-done or abandoned as a result of inadequate documentation.

Program design languages, with data flow diagrams, allow concise documentation *to be produced at low cost. Also this documentation* has direct relevance to designer, manager and customer.

The use of a program design language thus directly provides a kernel of the documentation as a side-product.

5.12. Visibility

Clear, indented, control structures illustrate flow of control.

The use of unrestricted - length symbols allow tight coupling between the model and the system being modeled.

Abstraction and module size limitations allow the reduction of each module's complexity to where the designer, and other people, can easily understand the operation of the modules.

The fact that there are no "run-time" costs in this form of design allows the clearest algorithms to be used, and information hiding can be used to extremes. During implementation simple optimisation techniques can be applied where necessary [Shooman].

The machine generated calling hierarchy, and cross references also show logic relations not normally visible during the design process.

Program design languages are a simple route to providing visibility at low effort.

5.13. Communication

Computer programming languages are used for communication. The most discerning recipient is the compiler. A compiler is extremely intolerant and has a very restricted capability. because of these facts, all effort is made to communicate effectively in terms of a compilers requirements. As a result, *communication to other people and the designer is prejudiced.*

Program design languages are also used for communication. However, the priorities are different.

The key audience is the designer. Designer's efficiency and effectiveness is directly related to the support that tools provide. A program design language

is tolerant, allowing every part defined in the design to be documented with minimum effort. The results can be tightened as required. The alternations of design diversification and convergence is not restricted by the language.

The visible format also makes this document easy to use for the designer.

The next priority in communication is to other people. Designs are easily made transparent, to provide maximum access for design reviews and walkthroughs.

The lowest priority is communication to a computer. As no directly executable code results, the designs need only be complete to the level required to illustrate the algorithms. From this point of view, program design languages can be regarded as Very High Level Languages.

The orientation of program design languages is to make the design part of the solution, and not of the problem. This is achieved by the highly visible, simple form, which maximises communication.

5.14. Early Error Removal

Through the act of preparing a design, and particularly by reviews and walkthroughs, a number of logic errors are found.

The indented format of control structures, coupled with package diagnostics show control flow construct errors.

The use of calling hierarchy trees, and calling and data cross references, show the structure of the design, and its degree of cohesion and coupling. Certain types of errors can clearly be seen - uncalled or "orphan" modules, modules called where not expected, naming clashes, spelling errors in module or data names, imbalanced hierarchies, and un-used or once-only used data items.

Many of these errors are not visible in other types of design tools.

Finally the ease of continuing the design process to its logical conclusion of no remaining design decisions produces complete designs. The syndrome of "any reasonable programmer should be able to design this code" to fore-shorten the design process can readily be avoided.

The designs can be presented in such a form that customer, end users and consultants can verify that implicit and explicit requirements are met. This can be done at any stage of the design refinement.

All these factors allow and encourage the early identification and removal of design errors. The earlier removal of errors reduces costs by an exponential rate. Confidence is high, leading to motivation of staff.

5.15. Checking Procedures

Machine supported packages, with a certain level of fixed syntax allows standardisation.

Standardisation of format and approach results in reduction of errors, and easier comprehension. The designs are readily checked for conformance by a cook-book check list (Shooman 1983, Fagan 1976).

The calling hierarchy provides an opportunity to do systematic checking of design structure.

Data indexes allow orthagonal data flow checking, by forcing particular data values, and verifying the operations of all modules accessing these data items. In a similar manner, initialisation - before - use can be checked for any level of module invocation.

These factors allow and encourage systematic error checking procedures to be built. These will maximise the effectiveness of review and walkthroughs.

5.16. Front-end Loading

In systems engineering, typically 90% of lifecycle cost is committed within 5 % of the system design (Sperrius, Symposium). By evaluating alternate forms of design early, it is possible to vary the total system cost.

Program design languages allow this evaluation with no commitment to particular hardware or software. To discard one approach for another has less financial and emotional costs.

The capability of refining the design of processes to any level allows the exploration of all critical areas. If timing tests are necessary, they can be coded in prototype form with little effort.

Finally, front-end loading has value to management by reducing perceived risk, and increasing confidence.

5.17. Productivity

Program design languages allow and support the use of structured techniques. The value of structured techniques has been mentioned above.

An encyclopediac assessment of factors affecting the cost (and schedule) of software is given by Boehm (1981).

A summary is given of factors affecting design, and design with program design languages in particular.

5.17.1. Modern Programming Practices Effort Multipliers

Phase:	Require-ments and Product Design	Detailed Design	Code and Unit Test	Integration and Test	Overall
Usage:					
Very Low	1,05	1,10	1,25	1,50	1,24
Low	1,00	1,05	1,10	1,20	1,10
Nominal	1,00	1,00	1,00	1,00	1,00
High	1,00	0,95	0,90	0,83	0,91
Very High	1,00	0,90	0,80	0,55	0,82

Table 3

Usage

Very Low : No use of modern programming practices

Low : Beginning, experimental use of some modern programming practices

Nominal : Reasonably experienced in use of some modern programming practices

High : Reasonably experienced in most of the modern programming practices

Very High : Routine use of all modern programming practices

5.17.2. Modern programming practices

- Top-down requirements analysis and design. Includes hierarchical elaboration

- Structured design *notations* such as program design languages, structure charts and HIPO

- Top-down incremental development

- Design and code walkthroughs

- Structured code

- *Program* librarian

Productivity Ranges of Individual Techniques

Structured programming	1,78
Design and code inspections	1,54
Top-down development	1,64
Chief programmer teams	1,86

Table 4

5.17.3. Software Tools Effort Multipliers

Phase:	Requirements and Product Design	Detailed Design	Code and Unit Test	Integration and Test	Overall
Usage:					
Very Low	1,02	1,05	1,35	1,45	1,24
Low	1,00	1,02	1,15	1,20	1,10
Nominal	1,00	1,00	1,00	1,00	1,00
High	0,98	0,95	0,90	0,85	0,91
Very High	0,95	0,90	0,80	0,70	0,83

Table 5

Usage:-

- Very Low : Basic microprocessor-level tools (assembler, breakpoint monitor)
- Low : Mini-level tools (compilers, macroassemblers, library aids, basic data base aids)
- Nominal : Multi-user operating system, interactive debuggers
- High : Virtual memory operating systems, data base design aids, simple program design language etc.
- Very High : Ada APSE level tools

These studies indicate the value of modern methodologies, and in particular the value of *program design languages*.

5.18. Reusability

A fundamental approach to reduce the *cost of software development* is the reuse of prior software. The reuse can be based on objects (libraries), source (source libraries), design (expressed in program design language or other methods) or documentation (algorithms described by text).

The use of a program design language to provide portable software is particularly successful. The algorithm is expressed in a concrete form, but *coding effort* is relatively trivial (Freeman 1983).

5.18.1. Chief Programmer Teams

This *organisational structure* is intended to minimise complex team interaction and maximise design results. A single "chief programmer" controls the whole project, and implements at least the major parts himself, with the aid of a dedicated support team. "The chief programmer is intended to be an extremely dedicated and competent person *achieving ten to twenty* times average output. Whilst this method has had tremendous successes, it has also had tremendous

failures. Results are very dependent on *individual performance*, and have proved to be risky (Yourdon 1979 p 160; Baker 1977, Brooks 1975, Boehm 1981).

One member of the support team is the program librarian. This role has had success outside this method *and is discussed* further.

5.18.2. Program Librarian

This position has the responsibility for configuration management of the project. Baselines are defined, changes and updates are controlled. Backups are performed, and versions are accepted, distributed and archived. Tests are monitored and recorded. Finally assistance may be provided with entry and documentation.

This function can readily be combined with any design methodologies and techniques. The results are higher output by *task specialization*, documentation control, and risk reduction (Yourdon 1979 p 175, Brooks 1975, Myers 1976).

5.18.3. Use of Program Design Languages with Data Flow Diagram

The role of data flow diagrams is to provide a macro representation of external entities, processes, data flows and data stores (Gane and Sarson 1979).

Program *design languages* are particularly useful to define behaviour of external entities and processes, in logical form, and the data structures associated with data flows and data stores.

The use of both data flow and program design using these methods is particularly effective. Both a global overview and concrete details can be represented (Walker 1986).

6. COMPILER TECHNIQUES

6.1. Introduction

6.1.1. Compilers

A compiler is a form of program translator. A translator is itself a program which translates from a source language to a target language. The translator itself is written in an implementation language. Various forms of translators existi-

- Assemblers
- Compilers
- Interpreters

An assembler translates from assembler mnemonics to machine code.

A compiler translates from a source language (usually a high-level language) to a target language (usually machine code).

An interpreter translates from a source language to perform the operations required directly.

A compiler is a batch-mode program. The source is prepared and made available in a typically in a file. The compiler is invoked, reads the source, and produces a listing and an object file of target language.

The listing file is used to provide object file information, related to the source information, with error messages, to the user.

The object file is used to run the program, and may be directly executable.

6.1.2. Relevance to Program Design Languages

A program design language has two visible forms : input and output. The input form can be regarded as a source language, and the output form can be regarded as a listing.

The listing has many similarities to a compiler's listing. Some compilers even provide similar indentation and referencing facilities. However, no object file is produced.

The use of formal source definitions is now standard practice. They allow concise definition of many of the aspects of the language. These definitions can be used directly to design the compiler.

Computer design theory is well established and documented. It is probably the most defined area of software engineering. The constructional aspects are also rigorously defined, with the exception of detailed code construction which is dependant on the target language's peculiarities.

The program design language input is similar to a source language to a

compiler, and the program design language formatter is a subset of a compiler, and the listing is used to present the formatted design.

6.2. Phases of a Compiler

The phases of a compiler are:-

- lexical analysis
- syntactic analysis
- semantic analysis
- code generation

The lexical analysis phase reads the source file to obtain a sequence of tokens. A token is a unit of input, typically an identifier, a number, an operator or a reserved word (such as "else"). These tokens are more easily handled by the rest of the compiler than its character form. The token type (identifier, number, operator etc.) and value (character string, number value, "addition") are made available to the rest of the compiler.

The syntactic analysis phase uses the tokens from the lexical analysis phase to analyse the form of the input. It recognises major parts such as data structures, procedures and constructs. During its operation it checks validity of the input, and builds internal tree forms representing the program's structure.

The semantic analysis phase checks interrelationships within the program structure. For example it may check that all procedures invoked are defined, that data types are used consistently, and that format and actual parameters match. Various parts of the internal tree forms are compared.

The code generation phase uses the internal representations to provide the object language output.

The listing file is produced in part by all of the phases. Error messages can be generated as a result of each phase.

In a normal compiler optimisation may be performed to reduce the object code size and increase its execution speed. This would occur both on the internal representation and the code.

For the design language formatter the optimisation and code generation phases do not exist. The sole output is a listing file.

These phases may take place sequentially, or may be interleaved. One approach may be to have a syntax analyser, calling lexical analyser fragments as required, followed by a semantic analyser calling code generator fragments. This would be classified as a two pass compiler.

A number of references are given on compilers. Particularly useful are Aho and Ullman (1978), Aho, Sethi and Ullman (1986), Bauer and Nickel

(1974), Lewis, Rosenkrantz and Stearns 1976, and Rohl (1975).

6.3. Formal Definitions

Programming languages were initially designed and defined in an ad-hoc manner. The definition was in the form of English definitions supplemented by examples of correct usage. Ambiguities, omissions and errors were very common. In practice, the language standard was what the compiler would accept. The need for a rigorous formal language definition method became clear.

Both mathematics and natural language definitions were studied to find solutions to this problem. The major breakthrough was the ALGOL 60 definition by Naur et al (1960). This used a simple re-writing method (Backus Normal Form) to define syntax. This method used a number of productions (or rules). Each production would indicate how a higher level abstraction (non-terminals) could be re-written in terms of concrete characters (terminals) and non-terminals. This method allowed at least the basic form of a language to be defined absolutely. Supplementary information in terms of semantics was provided in English, and by examples. In honour of Naur's efforts, the method is now also known as Backus Naur Form (BNF).

Further methods were evolved - Extended Backus Notation, Van Wijngaarden Forms (V-grammars), Production Systems, Vienna Definition Language and Attribute Grammars. A full definition of a language and its environment has also been achieved - this is a complete definition of syntax, semantics and run-time behaviour (Cleaveland and Uzgalis 1977).

The level and type of formal definition used is similar to BNF. Most references on compilers include additional information. However, complete references are Cleaveland and Uzgalis (1977), Marcotty and Ledgard (1976), and McGettrick (1980).

A similar approach to input definition can also be used for internal and output form representation.

6.4. Parsing Techniques

Parsing is the activity of transforming tokens into an internal representation and a symbol table. It is part of the syntax analysis phase.

Assuming the existence of a BNF type definition of the language, the program may be recognised by two main approaches:-

- up-down
- bottom-up

The top-down approach uses a sequence of goals, each goal representing terminals and non-terminals expected. The input is then scanned and a match is sought for terminals. The parse tree is thus built from the top and from the left as the input is recognised. A typical grammar is LL(1). This grammar allows scanning from the left to right of the source, and recognised from the left using a single symbol.

A grammar is LL(1) if all productions for the same non-terminal are *selectable by inspection of only the first* symbol. Most languages can have a grammar rewritten into this form. This form of grammar is particularly suitable for hand coding with a recursive descent compiler - where the program has procedures whose bodies represent the terminals and non-terminals of the grammar (Bauer and Rickel 1974, Lewis, Rosenkrantz and Stearns 1976, Waite and Goos 1984, Aho, Sethi and Ullman 1986).

The bottom-up approach starts with the input which is stacked until a set of terminals and non-terminals on the stack are recognised. These are then replaced by the non-terminal representing that production. This is continued until only the most abstract non-terminal remains on the stack, and the input file has been read. A typical grammar is LR(1). This grammar allows scanning from left to right, recognises fr the right of the stack, assisted by observing one symbol in the input.

It can be shown that LR(1) is a more powerful grammar that includes LL(1). However, although both grammars are easily used for practical languages, LR(1) is most suitable for automated parsers.

A wealth of references exist on grammars, including specialised forms. Good references include Aho, Sethi and Ullman (1986). Lewis, Rosenkrantz and Stearns 1976, Bauer and Rickel (1974), and Wai[illegible]3 (1984).

6.5. Representation

A parse tree may be built-up as the synta[illegible] s phase proceeds. Concurrently a symbol table may be built.

The tree is a linked-list representation, with nodes representing non-terminals and leaves representing terminals. Various contents can be provided - they may correspond directly to the grammar, or may represent only the minimum structures for later use. In fact a tree need often not be generated explicitly - the code may be emitted while the analysis is being performed.

The symbol table is used to contain the information gained about symbols. It includes the actual character strings, its type and attributes, and may include information about scope and usage.

An extended form of representation is also possible - to annotate a parse tree with all the information relating to the symbol table, as well as other information from semantic analysis. The cross-linking results in a linked network. Structure related symbol searches are possible, and in particular the generation of
recursion-free calling trees is possible.

Whilst not relevant to a program design language processor, the use of such "trees" as well as tree walking algorithms allows simple construction of compilers with accurate code generation.

The use of such representations is covered in Bornat (1979), as well as general compiler texts.

6.6. Selection

In this particular project issues of structured design and programming are regarded *as important*. *Good design* practice and compiler design practice are objectives. The use of table driven parsers, and automatic parsers (compiler-compilers) was rejected as not showing practices applicable to general software engineering problems.

The recursive descent parser was chosen. It requires a recursive implementation language, with its implicit stack mechanism. Examples of compilers written in this method are given in Welsh and McKeag (1980), Wirth (1976), and Zelkowitz, Shaw and Gannon (1979).

7. DESIGN OVERVIEW

7.1. Introduction

7.1.1. Purpose

This design overview is intended to be used as a supplement to the Program Design Language Processor User Manual and the Program Design Language listings.

7.1.2. Scope

The design document is intended to be read by a reasonably skilled programmer who has used the Processor extensively. The main philosophy and structure is defined in sufficient detail to allow easier understanding of the design. Familiarity with compiler design methods, formal grammars, BNF notation and use of recursive procedures on lists will also aid understanding.

7.1.3. Listings and References

The design listings are:

- the parser (PDL - PARSER in the file INITA.PDL, and related files defined in it)

- the formatter (FORMATTER - LISTER in the file GENIND.PDL, and related files defined in it)

- the parse tree structure definition (PARSE - TREE in the file TREEDEF.PDL)

- the intermediate file definition (PREPARED_FILE FORMAT in the file PREPDEF.PDL)

The program listings are:

- the menu program (in the PDL.PAS file)

- the parser program (INITA.PAS and related files defined in it)

- the formatter program (GENIND.PAS and related programs defined in it)

The references to the code are:

- Wirth, N "Algorithms and data structures" Prentice-Hall, 1976 (used for the balanced tree code).

- Jamsa, K; Namerotf, S "Turbo Pascal Programmers Library" Osbourne McGraw-Hill, 1986 (used for the menu package in the PDL.PAS program for the information input).

The specification for the input format and the output listings is the User Manual itself.

7.2. General Program Design

7.2.1. Introduction

The processor consists of three programs:

- the first program (PDL.COM) requests user information and executes the second program.

- the second program (INITA.COM) reads in the source, parses the source generating an internal tree, outputs an intermediate file and executes the third program.

- the third program (GENIND.COM) reads in the intermediate file and outputs the formatted design.

7.2.2. Design Approach

The design is based on building a comprehensive explicit parse tree in memory, then interpreting and augmenting the parse tree and then reading the tree in the required order to provide listing data.

The use of an explicit parse tree makes the program's actions highly visible and provides structural information allowing more detailed debugging, as well as a possible interface to executable computer language generators.

The parse tree is initially generated in a single pass from the source. Further passes are made to augment the tree with balanced binary trees of lexically ordered procedure and data indexes.

The text is fully cross referenced using and augmenting the binary trees.

The parse tree is further annotated by usage and calling relations, and line and page numbers.

No separate symbol tables are used - rather the tree is augmented by internal links and data. This allows the use of arbitrary scope rules as well as a backward and forward chaining of references.

The design is oriented to be as simple as possible within the constraints of the problem's complexity. Use is made of recursion in many cases. This provides simplicity because of the close match to the inherently recursive nature of the input syntax. The recursive procedures are matched one-to-one by similarly recursive data structures.

Beyond the use of these recursive techniques - which are applicable to many design requirements - no explicit use is made of specialist compiler construction techniques. This approach is specifically selected to illustrate general design procedures making extensive use of a program design language package.

The first stage in the design was to define the Program Design Language itself. The language is based on that defined by Caine and Gordon (1975) and used by the author and his team, with enhancements in numerous areas including

cross referencing within lines (thus including functions), the maximum size of design allowable, and the improved formatting by allowing multiple segments on one page.

The language was defined by writing a User's Manual with a formal syntax written in BNF, and providing full information on the logical presentation of the output listings.

Next the syntax was extended to include a complete error syntax. In other words all input - valid or invalid - was formally defined.

This was done by adding a default syntax entry to every set of valid alternatives, and by identifying all re-synchronisation tokens (Aho and Ullman 1979 etc.). This was simplified by the line-by-line and formatting related syntax of the language.

This error syntax formed part of the User's Manual, and related the error messages that would appear in the listings to the syntax itself.

The next procedure followed was to generate an LL - 1 grammar from the complete input syntax defined in the User's Manual. This is simply an unambiguous form which allows all alternatives to be selected from input already read.

Next a comprehensive output syntax, in logical form, was defined from examples of output listings and descriptions, and checked by considering each input in the language.

A full input parser was written to consume valid and invalid input. This input parser was based exactly on the grammar of the language - each non-terminal is represented by a procedure calling further procedures relating to the next non-terminals, and each terminal is represented by a routine to consume the terminal.

This design was extensively desk-checked, as any errors here would have a correspondingly greater effect during further stages of development.

An internal data representation was then defined. This data structure set consists of a data structure for each non-terminal of the language. Each data structure was then extended to hold all of the items required for the listing - for example page and line numbers, where used and where defined links for each procedure and each data item, lexically sorted trees of procedures and data items, and design statistics.

In effect all knowledge of the design is contained in the data structures and the links between them.

To allow tree walks to be performed, each data structure's type is defined, either by explicit tags, or implicitly by its links to higher level data structures.

The parser was then updated to consume input and generate the initial parse tree containing all the input text.

As input text is read corresponding to a new structure, the new structure is created, initialised, linked to the existing structures and filled with the input text.

Once the initial parse tree is created the input file is no longer required. Various tree-walks are performed to provide additional data. Each walk performs a separate function - no attempt was made to optimise performance by commoning walks at the expense of confusing the design logic.

The final parse tree includes a network of called and calling procedures linked together as well as a network of all data items. Page and line numbers also exist. All the indexes can be read by simple tree walks.

The intermediate file is generated from simple walks providing table of contents (sequential walk of groups and segments), formatted and cross referenced text (Sequential walks of groups, segments and text, including the references to defined procedures and noting the nesting levels), calling trees (a walk along the procedures used links, but terminating on cycles), procedure and data indexes (using lexical walks, and then using the where used data structures in sequence) and statistics.

The final listing formatter is a simple cosmetic routine following the top-down form of the output listing, reading the file to consume the input, and providing ASCII output for listing.

The formatter has no significant data structures.

The intermediate file which is read has a unique identifier for each line or part of a line allowing the formatter to interpret the input easily.

The formatter is an inverted program - instead of following the sequence of the intermediate file, it follows the logical sequence of the listing, section by section, page by page, and item by item, consuming the input in so doing (Jackson 1975).

The formatter provides headers, footers, titles predefined formats and the indentation - the cosmetics of the listing.

7.2.3. Program Structure

The program structure is tightly coupled to the recursive input syntax. A single procedure is defined for each function. This results in a recursive descent parser with no backtracking. Each production is matched by a data structure which is created and linked to the previous data structures. Pointers are extensively used.

The "var" or reference by address mode of parameter passing is extensively used. To add a structure the following steps are typical:

- the procedure is called, passing the address of the link area of the previous structure (initialised before to nil)

- the procedure determines whether a structure is to be created

- if so, a memory allocation function is called, returning the start of the area in the passed parameter location (i.e. the link which is now set)

- the procedure now initialises the new structure, with all links set to nil

- the procedure now calls all the next level procedures, in turn passing the address of the link areas in the new structure for each specific procedure

Note that recusion can be used with this method.

The "var" definitions may not be omitted - this results in incorrect links, which are typically only read in a completely different part of the code. The code would then crash with no obvious cause. This must be carefully desk-checked.

All Pascal procedures are written as non-nested procedures, unless specifically necessary - such as the balanced tree search and insert procedures where the lifetimes of the temporary variables are controlled.

A method of coordinated procedure calls and link address passing is also used for the tree walks,

Extensive use is made of variant records. This has necessitated the use of the GetMem and SizeOf functions of Turbo Pascal.

In order to facilitate initial debugging, each procedure is written so that the first executable line will list the procedure's name. This feature can easily be removed or re-installed using global edits.

As far as possible total compatibility of PDL and code is maintained.

All the indirectly recursive procedures have the forward declarations.

Use was made of symbolic constants.

Full-length symbol names were used to minimise the effort to understand the programs.

The naming of the data areas, pointers to the areas, and the dereferencing methods used were carefully coordinated.

Read-ahead was used to ensure a new line of text was always available.

The philosophy of the PDL.PAS program was completely different to the other programs - it is based totally on a published package that provides a system of prompts, read-in facilities, and temporary help lines.

The code for this program consists only of input to this package, validity checks on the paths and filenames given, the checking of program availability, and the generation of temporary command files, which are all totally MS-DOS specific.

7.2.4. Data Structures

All data structures are dynamically generated using GetMem and SizeOf functions of Turbo Pascal, and are either explicitly defined (by headers) or implicitly defined by content or context. No de-allocation of structures is performed. The data structures are given in a BNF-like form in the design listing.

All fixed size fields are part of the nodes, but the variable length strings are separate data items pointed to by the main structure.

The main structure is a parse tree formed from the linked consecutive groups in order of appearance with an initial default, as using groups is optional in the language.

Each group may have consecutive linked segments - procedure, data definition or comment segments.

Each segment consists of its constituent linked parts - the literal and data definition segments are simple sequences of lines, while the procedure segment is a nested set of sequence, selection and iteration blocks, with general strings, conditionals and comments.

The parse tree also has a lexically ordered balanced tree of procedures. This is created by adding links to the existing procedure structures.

A similar tree of data items is created. As the data items can be in any position within the text line, the data item is repeated in a single data item node and string set.

A set of "where used" and "procedures used" links is generated for the cross references. These consist of simple nodes.

Line and page number entries and links are also provided.

7.2.5. Implementation Requirements

A full parse tree is generated dynamically. This requires sufficient memory. Pointer operations are also required. Only a limited amount of static data storage is required.

A natural pass oriented breakdown is defined. This encourages the use of chained program fragments so no special requirements exist on code size (total code size was already predicted to be well over the limit of 64K bytes of some compilers).

All file read and write operations are purely sequential, requiring no special operations.

Finally the recursive nature of the design requires an implementation language with efficient and convenient facilities for recursion and stack-based local variables.

This support appears to be fully provided by Turbo Pascal version 3 on an IBM PC or compatible as code limits of 64k bytes need not be exceeded per program part, relatively small data areas are required, and a large heap is available for dynamically generated data. The processing of the design itself would require a full complement of RAM. Most designs would need much less RAM.

7.3. Detailed Description

The detailed design is illustrated by examples taken from the full design listings. They are simplified actual text from the design. Some formatting changes had to be made to allow the examples to fit the width of the paper. A full description is given in the procedures in the full listing. Many have been removed here because they would duplicate the text, and also because of differences in the paper width would require changing their contents. Certain of the concepts require prior understanding of the language and its processor, obtainable from the User Manual.

7.3.1. Parse Tree Description

The parse tree consists of a number of data structures which are all similar in concept. Some main structures are given to illustrate the methods used.

Parse tree

The top level structure is:

```
1B      1  PARSE_TREE
2A      2  .  TITLE
2D      5  .  REVISION_
2E      6  .  TIME_
3C      9  .  DATE_
4A     13  .  GROUP
15E   138  .  PROCEDURE_SEGMENT_TREE
16A   140  .  DEFINITION_TREE
16B   142  .  STATISTICS_
```

Items 2 to 9 relate to headings.

From item 13 all groups are linked.

Item 138 is a pointer to the balanced tree of all procedures. It points to the start of the "middle" procedure. The left and right pointers should then be followed to any other procedure.

Similarly item 140 points to the "middle" data item node.

Item 142 is the summary of the statistics printed at the end of the designs.

The parse tree is illustrated below by the actual design entry.

Note the conventions used for the poinetrs and the data areas. The calling tree only references the data areas.

PARSE_TREE

```
-------------F-   1B  -F-----------------------------------
```

```
      P  1   ...This is the top level node for the entire
                design.The main parts
      P  2   ...are:
      P  3   ...top level parameters
                (TITLE_,REVISION_,TIME_ and DATE_)
      P  4   ...the starting node for all GROUP_'s
      P  5   ...the starting node for the
                PROCEDURE_SEGMENT's and
      P  6   ...the DEFINITION_'s in alphabetically
                ordered binary trees
      P  7   ...and STATISTIC_'s.
   2A P  8     TITLE_
   2D P  9     REVISION_STRING_POINTER >> REVISION_
   2E P 10     TIME_
   3B P 11     DATE_
   3F P 12     GROUP_POINTER >> GROUP_
  15A P 13     PROCEDURE_SEGMENT_TREE_POINTER >>
                  PROCEDURE_SEGMENT_TREE
  15B P 14     DEFINITION_TREE_POINTER >> DEFINITION_TREE
  15C P 15     STATISTICS_
```

The parse tree is now illustrated by the Pascal type definition:

```
     PARSE_TREE =
        record

        FIRST_TITLE_STRING_POINTER      : STRING_POINTER_TYPE ;
        REST_OF_TITLE_STRING_POINTER    : STRING_POINTER_TYPE ;
        REVISION_STRING_POINTER         : STRING_POINTER_TYPE ;
        HOUR_                           : HOUR_   ;
        MINUTE_                         : MINUTE_ ;
        DAY_                            : DAY_    ;
        MONTH_                          : MONTH_  ;
        YEAR_                           : YEAR_   ;
        GROUP_POINTER                   : GROUP_POINTER_TYPE ;
        PROCEDURE_SEGMENT_TREE_POINTER: SEGMENT_POINTER_TYPE;
        DEFINITION_TREE_POINTER         :
                DEFINITION_POINTER_TYPE;
        PROCEDURE_COUNT                 : COUNT_  ;
        PROCEDURE_LINE_COUNT            : COUNT_  ;
        DATA_ITEM_COUNT                 : COUNT_  ;
        DATA_ITEM_USAGE                 : COUNT_  ;
        ERROR_COUNT                     : COUNT_

        end;
```

Group

```
    4A      13  .  GROUP_
    4A      14  .  .  GROUP_ (    13 )
```

```
    4B      15  .  .  FULL_PAGE
    4C      16  .  .  GROUP_TITLE
    4D      17  .  .  SEGMENT_
```

The group structure consists of a pointer to the next group in sequence, the page number (whole number only, no part), a pointer to the title, and a pointer to all segments in the group.

The group definition from the design is given below.

GROUP_

```
-------------P-   3P  -P-----------------------------------------

       P  1    ...This is a node for GROUP_'s linked in lexical
                  order.
       P  2    ...The node gives the first PAGE_ and TITLE_
                  of the GROUP_;
       P  3    ...and all the lexically ordered SEGMENT_'s in
                  the GROUP_.
    3F P  4      GROUP_POINTER >> GROUP_
    4A P  5      FULL_PAGE
    4B P  6      GROUP_TITLE_POINTER >> GROUP_TITLE  o
    4C P  7      SEGMENT_POINTER >> SEGMENT_
```

This is the corresponding Pascal type definition:

```
{
      This is a node for GROUP_'s linked in lexical order.
      The node gives the first PAGE_ and the TITLE_ of the GROUP_,
      and all the lexically ordered SEGMENT_'s in the GROUP_.
}

      GROUP_ =
          record

          GROUP_POINTER                 : GROUP_POINTER_TYPE ;
          FULL_PAGE                     : FULL_PAGE ;
          GROUP_TITLE_POINTER           : STRING_POINTER_TYPE ;
          SEGMENT_POINTER               : SEGMENT_POINTER_TYPE

          end;
```

Segment

```
    4D      17  .  .  SEGMENT_
    5A      18  .  .  .  PROCEDURE_SEGMENT
    13A     104 .  .  .  DATA_DEFINITION_SEGMENT
    15A     128 .  .  .  LITERAL_SEGMENT
```

These are all the types of segments. Only the procedure segment is considered further. Below the design entry is given.

```
SEGMENT_

-------------P-   4C  -P-----------------------------------------

      P  1    ...All types of SEGMENT_'s.
   5A P  2      PROCEDURE_SEGMENT
      P  3    ...or
  12C P  4      DATA_DEFINITION_SEGMENT
      P  5    ...or
  14B P  6      LITERAL_SEGMENT
```

This is the Pascal definition:

```
{
      SEGMENT_'s can be of the following types:
}
SEGMENT_TYPE =
      (

          PROCEDURE_ ,
          DATA_DEFINITION ,
          LITERAL_

      ) ;
```

Procedure Segment

```
   5A      18  . . .  PROCEDURE_SEGMENT
   4D      19  . . .  . SEGMENT_(    17 )
   5B      20  . . .  . PAGE_PART
   5C      21  . . .  . HIGHEST_LINE_NUMBER
   6A      22  . . .  . SEGMENT_TYPE
   5A      23  . . .  . PROCEDURE_SEGMENT (    18 )
   5A      24  . . .  . PROCEDURE_SEGMENT (    18 )
   6B      25  . . .  . BALANCE
   6C      26  . . .  . PROCEDURE_NAME
   6D      27  . . .  . PARAMETER_
   6E      28  . . .  . REST_OF_LINE
   6F      29  . . .  . WHERE_USED
   7B      33  . . .  . LINE_NUMBER_IN_TREE
   7C      34  . . .  . PROCEDURE_USED
   7A      37  . . .  . LINE_NUMBER (    32 )
   7D      38  . . .  . CONSTRUCT_
```

This is examined in more detail.

Item 19 points to the next segment in sequence.

Item 20 is the page and the part of the page where the procedure is printed.

The highest line number is that of the final line in this procedure. It is used to determine if this procedure can still fit on the current page.

The segment type is "PROCEDURE_SEGMENT".

Items 23, 24 and 25 relate to the balanced binary tree of the procedures. The first points to the "middle" procedure to the left of the current, and the other to the right. A depth-first, left to right walk will produce all the procedures in lexical order. The balance is used to indicate whether this node has a longer tree on the left or right or whether it is balanced.

Items 26, 27 and 28 are in the title of the procedure definition.

WHERE_USED points to the nodes which indicate the procedure in which the current procedure is used, the line number, and the next WHERE_USED in lexical order.

Item 33 is the position in the calling tree (i.e. the LINE_NUMBER_IN_TREE defined on page 7B is itself number 33 in the parse tree calling tree).

Item 37 is the line number of the procedure title - always set to "0". This method is used for consistency with other line numbers for procedure and data cross referencing.

Finally item 38 points to the first construct of the procedure.

The design entry is given below.

```
PROCEDURE_SEGMENT

------------P-   5A  -P-----------------------------------------

    4C P 14      SEGMENT_POINTER >> SEGMENT_
    5B P 15      PAGE_PART
    5C P 16      HIGHEST_LINE_NUMBER
    5D P 17      PROCEDURE_SEGMENT_TYPE >> SEGMENT_TYPE
    5A P 18      LEFT_PROCEDURE_POINTER >> PROCEDURE_SEGMENT
    5A P 19      RIGHT_PROCEDURE_POINTER >> PROCEDURE_SEGMENT
    6A P 20      BALANCE
    6B P 21      PROCEDURE_NAME_POINTER >> PROCEDURE_NAME
    6C P 22      PARAMETER_POINTER >> PARAMETER
    6D P 23      REST_OF_LINE_POINTER >> REST_OF_LINE
    6E P 24      WHERE_USED_POINTER >> WHERE_USED
       P 25   ...not used yet
       P 26   ...LAST_WHERE_USED_POINTER
    7A P 27      LINE_NUMBER_IN_TREE
    7B P 28      PROCEDURE_USED_POINTER >> PROCEDURE_USED
    6F P 29      LINE_0 >> LINE_NUMBER
    7C P 30      CONSTRUCT_POINTER >> CONSTRUCT_
```

The Pascal equivalent is:

```
PROCEDURE_SEGMENT_TYPE =
record

        SEGMENT_POINTER             : SEGMENT_POINTER_TYPE;
        PAGE_PART                   : PAGE_PART_TYPE ;
        HIGHEST_LINE_NUMBER         : LINE_NUMBER ;
        SEGMENT_TYPE                : SEGMENT_TYPE ;
        LEFT_PROCEDURE_POINTER      : SEGMENT_POINTER_TYPE;
        RIGHT_PROCEDURE_POINTER     : SEGMENT_POINTER_TYPE;
        BALANCE                     : BALANCE  ;
        PROCEDURE_NAME_POINTER      : STRING_POINTER_TYPE ;
        PARAMETER_POINTER           : STRING_POINTER_TYPE ;
        REST_OF_LINE_POINTER        : STRING_POINTER_TYPE ;
        WHERE_USED_POINTER          : WHERE_USED_POINTER_TYPE;
        LINE_NUMBER_IN_TREE         : LINE_NUMBER_IN_TREE ;
        PROCEDURE_USED_POINTER      : PROCEDURE_USED_POINTER_TYPE ;
        LINE_O                      : LINE_NUMBER ;
        CONSTRUCT_POINTER           : CONSTRUCT_POINTER_TYPE

end ;
```

Construct

```
 7D      38 . . . . . CONSTRUCT
 8A      39 . . . . . COMMENT_CONSTRUCT
 8D      44 . . . . . IF_CONSTRUCT
 10A     62 . . . . . WHILE_CONSTRUCT
 10C     69 . . . . . UNTIL_CONSTRUCT
 11A     76 . . . . . BLOCK_CONSTRUCT
 11D     84 . . . . . GENERAL_STRING_CONSTRUCT
 12A     90 . . . . . CASE_CONSTRUCT
```

These are all the types of constructs. Below the design entry is given. Only the if construct is examined further.

CONSTRUCT_

```
-------------P-  7C  -P-----------------------------
```

```
      P  1   ...All types of CONSTRUCT_'s.
 7D   P  2     COMMENT_CONSTRUCT
      P  3   ...or
 8C   P  4     IF_CONSTRUCT
      P  5   ...or
 9E   P  6     WHILE_CONSTRUCT
      P  7   ...or
 10A  P  8     UNTIL_CONSTRUCT
      P  9   ...or
 10C  P 10     BLOCK_CONSTRUCT
      P 11   ...or
 11A  P 12     GENERAL_STRING_CONSTRUCT
      P 13   ...or
```

If Construct

```
    8D      44  . . . . . IF_CONSTRUCT
    7D      45  . . . . . . CONSTRUCT_ (    38 )
    8B      46  . . . . . . CONSTRUCT_TYPE (    41 )
    9A      47  . . . . . . CONDITIONAL_
    7D      52  . . . . . . CONSTRUCT_ (    38 )
    9C      53  . . . . . . ELSEIF_
    7A      57  . . . . . . LINE_NUMBER (    32 )
    9D      58  . . . . . . ELSE_LINE_STRING
    7D      59  . . . . . . CONSTRUCT_ (    38 )
    7A      60  . . . . . . LINE_NUMBER (    32 )
    9E      61  . . . . . . ENDIF_LINE_STRING
```

Item 45 points to the next construct of this procedure.

Item 46 is set to IF_CONSTRUCT.

Item 47 is a pointer to the conditional associated with the if.

Item 52 is the "then" construct pointer.

Item 53 is the pointer to the optional elseif.

Item 58 is the optional else line - which must not have a further string, but if one exists, an error message is appended.

Item 59 links to the actual constructs of the else, and item 57 is the line number of the else.

Items 60 and 61 are for the endif line. This line must exist, and must have no further text. If it does not exist, one is generated, and an error message given. If further text exists, it is followed by an error message.

The if design is illustrated below.

IF_CONSTRUCT

```
------------P-   8C  -P---------- ---------------------------

    7C P  8     CONSTRUCT_POINTER >> CONSTRUCT_
    8A P  9     IF_CONSTRUCT_TYPE >> CONSTRUCT_TYPE
    8D P 10     CONDITIONAL_POINTER >> CONDITIONAL_
    7C P 11     THEN_CONSTRUCT_POINTER >> CONSTRUCT_
    9B P 12     ELSEIF_POINTER >> ELSEIF_
    6F P 13     ELSE_LINE_NUMBER >> LINE_NUMBER
    9C P 14     ELSE_LINE_STRING_POINTER >> ELSE_LINE_STRING
    7C P 15     ELSE_CONSTRUCT_POINTER >> CONSTRUCT_
    6F P 16     ENDIF_LINE_NUMBER >> LINE_NUMBER
    9D P 17     ENDIF_LINE_STRING_POINTER >> ENDIF_LINE_STRING
```

The Pascal if type definition is:

```
IF_CONSTRUCT_TYPE =
      record

          CONSTRUCT_POINTER                : CONSTRUCT_POINTER_TYPR ;
          CONSTRUCT_TYPE                   : CONSTRUCT_TYPE ;
          CONDITIONAL_POINTER              : CONDITIONAL_POINTER_TYPE ;
          THEN_CONSTRUCT_POINTER           : CONSTRUCT_POINTER_TYPE ;
          ELSEIF_POINTER                   : ELSEIF_POINTER_TYPE ;
          ELSE_LINE_NUMBER                 : LINE_NUMBER ;
          ELSE_LINE_STRING_POINTER         : STRING_POINTER_TYPE ;
          ELSE_CONSTRUCT_POINTER           : CONSTRUCT_POINTER_TYPE ;
          ENDIF_LINE_NUMBER                : LINE_NUMBER ;
          ENDIF_LINE_STRING_POINTER        : STRING_POINTER_TYPE

      end;
```

Conditional

```
    9A      47  . . . .  CONDITIONAL_
    9A      48  . . . .  . CONDITIONAL_ (    47 )
    7A      49  . . . .  . LINE_NUMBER (     32 )
    7C      50  . . . .  . PROCEDURE_USED (  34 )
    9B      51  . . . .  . ACTUAL_STRING
```

The conditional (c.f. "expression" in Pascal) in item 48 points to the next conditional (i.e. a multiple line conditional), followed by the line on which it is to be found, a pointer to which procedure is used, if any, and a pointer to the actual string.

The conditional design is given below.

```
CONDITIONAL_

------------P-  8D  -P-----------------------------------------

    6D P  5     CONDITIONAL_POINTER >> CONDITIONAL_ °
    6F P  6     LINE_NUMBER
    7B P  7     PROCEDURE_USED_POINTER >> PROCEDURE_USED
    9A P  8     ACTUAL_STRING_POINTER >> ACTUAL_STRING
```

The Pascal type definition is:

```
CONDITIONAL_ =
      record

          CONDITIONAL_POINTER          : CONDITIONAL_POINTER_TYPE ;
          LINE_NUMBER                  : LINE_NUMBER ;
          PROCEDURE_USED_POINTER       : PROCEDURE_USED_POINTER_TYPE ;
          ACTUAL_STRING_POINTER        : STRING_POINTER_TYPE

      end;
```

Elseif

```
    9C      53  . . . . ELSEIF_
    9C      54  . . . . . ELSEIF_ (    53 )
    9A      55  . . . . . CONDITIONAL_ (    47 )
    7D      56  . . . . . CONSTRUCT_ (    38 )
```

Similar to the conditional above, item 54 points to the next elseif, if any, the associated conditional, and the first construct.

The elseif design entry is given below.

ELSEIF_

```
------------P-   9E  -P-----------------------------------------

      F  1   ...This node is linked to each successive
                 ELSEIF_ in lexical order
      P  2   ...It is part of the IF_CONSTRUCT.
   9B P  3     ELSEIF_POINTER >> ELSEIF_
   8D P  4     CONDITIONAL_POINTER >> CONDITIONAL_
   7C P  5     CONSTRUCT_POINTER >> CONSTRUCT_
```

The Pascal is:

```
ELSEIF_ =
     record

         ELSEIF_POINTER              : ELSEIF_POINTER_TYPE ;
         CONDITIONAL_POINTER         : CONDITIONAL_POINTER_TYPE ;
         CONSTRUCT_POINTER           : CONSTRUCT_POINTER_TYPE ;

     end;
```

A full parse tree definition is given in the listing.

Data structures which do not relate directly to the input syntax but rather to the listing requirements are:

WHERE_USED (used for cross referencing)
PROCEDURE_USED (used for calling trees)
DEFINITION_ (used for explicit and implicit data definitions)

The internal data structure is a complete syntactically correct structure - all input errors are adjusted to provide full structures. For example a missing "ENDIF" in the input is remedied.

7.3.2. Input Definition

The input syntax is fully defined in the User Manual, both in BNF and by descriptions and examples of their use.

Some examples are:

```
17a )  ( text ) ::= ( comment segment )
   b )              ( procedure segment )
   c )              ( data segment )
   d )              ( group )

   24 )  ( procedure segment )
         ::=
         XP ( procedure line )[( procedure body )]

   27 )  ( procedure body ) ::= ( constructs )

29a )  ( constructs ) ::= ( construct )( constructs )
   b )                    ( construct )

30a )  ( construct ) ::= ( comment construct )
   b )                   ( if construct )
   c )                   ( while construct )
   d )                   ( until construct )
   e )                   ( case construct )
   f )                   ( block construct )
   g )                   ( procedure construct )
   h )                   ( general string construct )

   32 )  ( if construct ) ::= ( if line )( if body )( endif line )

   33 )  ( if line ) ::= IF ( conditional line )

   37 )  ( if body ) ::= [( then part )][( elseif parts )]
         [( else part )]

   38 )  ( then part ) ::= ( constructs )

39a )  ( elseif parts ) ::= ( elseif part )( elseif parts )
   b )                      ( elseif part )

   40 )  ( elseif part ) ::= ( elseif line )[( constructs )]

   41 )  ( elseif line ) ::= ELSEIF ( conditional line )

   42 )  ( else part ) ::= ( else line )[( constructs )]

   43 )  ( else line ) ::= ELSE ( end of line )

   44 )  ( endif line ) ::= ENDIF ( end of line )
```

7.3.3. Total System Structure

The simplified calling tree gives a picture of the whole system.

```
2B      1  FORMATTER
3A      2  . GENERATE_PARSE_TREE
3B      3  . . GENERATE_INITIAL_PARSE_TREE
3C      4  . . . INITIALISE
5B     12  . . . PARSE_TITLE
10B    48  . . . PARSE_REVISION
12B    55  . . . PARSE_TEXT_BLOCK
13A    58  . . . . PARSE_SEGMENT_
13B    61  . . . . . PARSE_PROCEDURE_SEGMENT_TAIL
14B    68  . . . . . . PARSE_CONSTRUCT_
14C    72  . . . . . . . PARSE_COMMENT_CONSTRUCT_TAIL
15B    79  . . . . . . . PARSE_IFTAIL
14B    93  . . . . . . . . PARSE_CONSTRUCT_
17A    94  . . . . . . . . PARSE_ELSEIF_
17C   102  . . . . . . . . PARSE_ELSE_
18A   111  . . . . . . . . PARSE_ENDIF_
14B   120  . . . . . . . . PARSE_CONSTRUCT_
18C   122  . . . . . . . PARSE_WHILETAIL
19C   139  . . . . . . . PARSE_REPEATTAIL
20C   158  . . . . . . . PARSE_CASETAIL
23A   194  . . . . . . . PARSE_BEGINTAIL
24C   217  . . . . . . . PARSE_GENERAL_STRING_CONSTRUCT
14B   254  . . . . . . PARSE_CONSTRUCT_
13A   255  . . . . . . PARSE_SEGMENT_ (   58 )
28B   257  . . . . . PARSE_DATA_DEFINITION_SEGMENT_TAIL
33A   309  . . . . . PARSE_LITERAL_SEGMENT_TAIL
34B   329  . . . . . PARSE_ASSUMED_LITERAL_SEGMENT
34C   336  . . . . . PARSE_UNIDENTIFIED_SEGMENT
35A   341  . . . . PARSE_GROUP_
12C   345  . . . . . CREATE_GROUP_ (    56 )
13A   350  . . . . . PARSE_SEGMENT_ (   58 )
35A   351  . . . . . PARSE_GROUP_ (   341 )
37B   353  . . GENERATE_PROCEDURE_SEGMENT_TREE
37C   354  . . . SEARCH_GROUP_FOR_PROCEDURE_NAME
38A   355  . . . . SEARCH_SEGMENT_FOR_PROCEDURE_NAME
38B   356  . . . . . PROCESS_PROCEDURE_NAME
38C   357  . . . . . . FIND_POSITION_FOR
                          NEW_PROCEDURE_SEGMENT
38A   376  . . . . . SEARCH_SEGMENT_FOR
                          PROCEDURE_NAME (   355 )
37C   377  . . . . SEARCH_GROUP_FOR_PROCEDURE_NAME
44B   378  . . GENERATE_EXPLICITLY_DEFINED
                    DEFINITION_TREE
44C   404  . . . . SEARCH_GROUP_FOR_EXPLICIT
                          DATA_ITEM (   379 )
51B   405  . . CROSS_REFERENCE
51C   406  . . . CROSS_REFERENCE_GROUP_
52A   407  . . . . CROSS_REFERENCE_SEGMENT_
52B   408  . . . . . CROSS_REFERENCE_PROCEDURE_SEGMENT
52A   471  . . . . . CROSS_REFERENCE_SEGMENT_
```

```
51C    472  .  .  .  .  CROSS REFERENCE GROUP  (   406 )
61B    473  .  .  GENERATE LINE AND PAGE NUMBERS
71B    522  .  PREPARE LISTING
71C    523  .  .  PREPARE FRONT_PAGE
72B    529  .  .  PREPARE TABLE_OF_CONTENTS
74C    536  .  .  PREPARE TEXT
74D    537  .  .  .  PREPARE GROUP
75A    538  .  .  .  .  PREPARE SEGMENT
75B    539  .  .  .  .  .  .  PREPARE PROCEDURE_SEGMENT
83A    587  .  .  PREPARE CALLING_TREE
86C    606  .  .  PREPARE PROCEDURE_INDEX
87C    613  .  .  PREPARE DATA_INDEX
88B    618  .  .  PREPARE LAST_PAGE
```

The design below is of the total system. Note that generate listing procedure has no reference - it is partitioned into another design document.

FORMATTER

```
------------P-   2B  -P-----------------------------------------

   3A P  8     Generate PARSE_TREE
  71B P  9     Prepare listing
      P 10     Generate listing
```

This is the procedure that creates the full parse tree.

GENERATE PARSE_TREE

```
------------P-   3A  -P-----------------------------------------

   3B P 10     Generate initial PARSE_TREE
  37B P 11     Generate PROCEDURE_SEGMENT_TREE
  44B P 12     Generate explicitly defined DEFINITION_TREE
  51B P 13     Cross reference
  61B P 14     Generate line and page numbers
```

The actual Pascal is given below. Note that additional debugging diagnostics were added, and retained afterwards to provide user feedback - one of the few User Manual changes in the development.:

```
Procedure GeneratePARSE_TREE ;
begin
{XWriteLn( 'GeneratePARSE_TREE' ) ; }

WriteLn( 'Reading in text from file - note disc accesses.' ) ;
GenerateInitialPARSE_TREE ;
WriteLn( 'Checking text for all procedure segments.' ) ;
GeneratePROCEDURE_SEGMENT_TREE ;
WriteLn(
     'Scanning the text for the explicitly defined data items.' ) ;
GenerateExplicitlyDefinedDEFINITION_TREE ;
WriteLn(
```

```
    'Cross-referencing all procedure invocations and data usages.'
    ) ;
CrossReference ;
WriteLn( 'Line and page numbers added.' ) ;
GenerateLineAndPageNumbers

end ;
```

The files associated with the parser design are given below.

INITA FILE

```
.------------C-   2A  -C-----------------------------------------

      C  1
      C  2      This part of the design is in file INITA.PDL.
      C  3
      C  4      The full set of files for the PARSER are:
      C  5      INITA PDL
      C  6      INITEXT PDL
      C  7      INITC PDL
      C  8      PROTREE PDL
      C  9      EXPDTREE PDL
      C 10      CROSSREF PDL
      C 11      NUMBERS PDL
      C 12      PREPROC PDL
      C 13      PREREST PDL
      C 14
      C 15      This listing should be used in conjunction
                  with those of
      C 16
      C 17      TREEDEF.PDL ( parse tree definition ),
      C 18      PREPDEF.PDL ( prepared file definition ), and
      C 19      GENIND.PDL ( formatter design ).
      C 20
      C 21      An overview of this design can be obtained by
                  examining the
      C 22      calling tree.
      C 23
```

7.3.4. Generate Initial Parse Tree

This is a recursive descent pure tree generator. It provides the creation and initialisation of most nodes. The input text is stored directly (retaining upper and lower cases), but with white space removed.

A part of the design is examined in more detail. The parse text block procedure consumes the actual design source. It first creates a default group (groups are optional in the language), then calls a procedure to look for all consecutive segments, and finally looks for all subsequent groups.

PARSE TEXT BLOCK

```
-------------P-  12B  -P-----------------------------------------

       P  1    ...Initial default GROUP_ - will be left empty
                     if followed directly by a group
   12C P  2      Create GROUP_(GROUP_POINTER)
   13A P  3      Parse SEGMENT_(SEGMENT_POINTER)
   35A P  4      Parse GROUP_(GROUP_POINTER)
```

The corresponding Pascal is:

```
Procedure ParseTextBlock ;
begin
{XWriteLn( 'ParseTextBlock' ) ; }

     {
         This is the main parsing procedure - GROUP_'s and
         SEGMENT_'s are parsed.
     }

     with PARSE_TREE_POINTER^ do
     begin
         {
         Initial default GROUP_ - will be left empty if
         followed directly by a group.
         }
         CreateGROUP_( GROUP_POINTER ) ;
         ParseSEGMENT_( GROUP_POINTER^.SEGMENT_POINTER ) ;
         ParseGROUP_( GROUP_POINTER^.GROUP_POINTER )
     end

end ;
```

This is a typical create routine:

```
CREATE GROUP_(ATTACHMENT_POINTER)

-------------P-  12C  -P-----------------------------------------

    4B P  1      Set ATTACHMENT_POINTER to
                    Allocate space(Size(GROUP_))
       P  2      Set GROUP_POINTER to NULL_
       P  3      Set FULL_PAGE to 0
       P  4      Set GROUP_TITLE_POINTER to NULL_STRING_POINTER
       P  5      Set SEGMENT_POINTER to NULL_
```

The Pascal equivalent is:

```
Procedure CreateGROUP_( var ATTACHMENT_POINTER :
    GROUP_POINTER_TYPE ) ;
begin
{XWriteLn( 'CreateGROUP_' ) ; }

GetMem( ATTACHMENT_POINTER , SizeOf( ATTACHMENT_POINTER^ ) ) ;
with ATTACHMENT_POINTER^ do
begin
     GROUP_POINTER := nil ;
     FULL_PAGE := 0 ;
     GROUP_TITLE_POINTER := NULL_STRING_POINTER ;
     SEGMENT_POINTER := nil
end

end ;
```

This routine looks for a segment. It does this by recognising the valid segment starts, processing the segments, handling the errors or by returning if the next group or end of text is found.

```
PARSE SEGMENT_(ATTACHMENT_POINTER)

-------------P-  13A  -P-----------------------------------------

      P  1   ...Parses all SEGMENT_'s in a GROUP_.
      P  2   ...Each routine recursively calls this routine.
      F  3   ...Note that faulty text is converted
                    into LITERAL_SEGMENT's.
   5C P  4     Remove blank lines
    6A P  5     if Symbol(XP)
   13B P  6        Parse PROCEDURE_SEGMENT tail
    6A P  7     elseif Symbol(XD)
   28B P  8        Parse DATA_DEFINITION_SEGMENT tail
    6A P  9     elseif Symbol(XC)
   33A P 10        Parse LITERAL_SEGMENT tail
    6A P 11     elseif not Symbol(XG)
    6A P 12        if Symbol(X)
    7D P 13           Accept symbol
   34B P 14           Parse Assumed LITERAL_SEGMENT
    6A P 15        elseif not Symbol(EOF_)
   34C P 16           Parse Unidentified segment
      P 17        else
      P 18          ...Group found - no action
      P 19        endif
      P 20     else
      P 21       ...no SEGMENT_ found, - no action
      P 22     endif
```

Here a procedure has been found. The procedure line is handled (including errors), the constructs are parsed, and the segment parse routine is recursively called.

```
PARSE PROCEDURE_SEGMENT_TAIL

-------------P-  13B  -P-----------------------------------------

       P  1    ...Expect a PROCEDURE_NAME followed by
                      possible CONSTRUCT_'s.
       P  2    ...move over the "%P".
     7D P  3      Accept symbol
     14A P  4     Create PROCEDURE_SEGMENT(ATTACHMENT_POINTER)
      5D P  5     if REST_OF_LINE blank
      7C P  6        Accept REST_OF_LINE
      5A P  7        Set PROCEDURE_NAME_POINTER to
                        Allocate string
                     (*** Missing PROCEDURE SEGMENT NAME)
       P  8          Locate PARSE_TREE
       P  9          Increment ERROR_COUNT
     14B P 10        Parse CONSTRUCT_(CONSTRUCT_POINTER)
       P 11      else
     26B P 12        Transfer PROCEDURE_LINE
     14B P 13        Parse CONSTRUCT_(CONSTRUCT_POINTER)
       P 14      endif
     13A P 15    Parse SEGMENT_(SEGMENT_POINTER)

Here all constructs are recognised.

PARSE CONSTRUCT_(ATTACHMENT_POINTER)

-------------P-  14B  -P-----------------------------------------

       P  1    ...Each routine recursively calls this
                      routine to parse
       P  2    ...all CONSTRUCT_'s in this SEGMENT_.
      5C P  3    Remove blank lines
      6B P  4    if Part Symbol(...)
     14C P  5       Parse COMMENT_CONSTRUCT tail
      6A P  6    elseif Symbol(IF)
     15B P  7       Parse Iftail
      6A P  8    elseif Symbol(WHILE)
     18C P  9       Parse Whiletail
      6A P 10    elseif Symbol(REPEAT)
     19C P 11       Parse Repeattail
      6A P 12    elseif Symbol(CASE)
     20C P 13       Parse Casetail
      6A P 14    elseif Symbol(BEGIN)
     23A P 15       Parse Begintail
     24B P 16    elseif not CONSTRUCT_ follower
     24C P 17       Parse GENERAL_STRING_CONSTRUCT
       P 18      else
       P 19         ...no CONSTRUCT_ found - no action
       P 20      endif
```

This is a typical construct parse routine. A separate procedure exists for each part of the construct.

Finally it calls the parse construct routine recursively.

```
PARSE IFTAIL

------------P-  15B  -P-----------------------------------------

       P  1    ...move over "if"
     7D P  2    Accept symbol
    15C P  3    Create IF_CONSTRUCT(ATTACHMENT_POINTER)
    16A P  4    Parse CONDITIONAL_(CONDITIONAL_POINTER)
    14B P  5    Parse CONSTRUCT_(THEN_CONSTRUCT_POINTER)
    17A P  6    Parse ELSEIF_(ELSEIF_POINTER)
    17C P  7    Parse ELSE_
    18A P  8    Parse ENDIF
    14B P  9    Parse CONSTRUCT_(CONSTRUCT_POINTER)
```

The Pascal is given below.

```
Procedure ParseIftail( var ATTACHMENT_POINTER :
    CONSTRUCT_POINTER_TYPE ) ;
begin
{XWriteLn( 'ParseIftail' ) ; }

{
      Move over 'if'.
}

AcceptSymbol ;
CreateIF_CONSTRUCT( ATTACHMENT_POINTER ) ;
with ATTACHMENT_POINTER^.IF_ do
begin
      ParseCONDITIONAL_( CONDITIONAL_POINTER ) ;
      ParseCONSTRUCT_( THEN_CONSTRUCT_POINTER ) ;
      ParseELSEIF_( ELSEIF_POINTER ) ;
      ParseELSE_( ATTACHMENT_POINTER ) ;
      ParseENDIF_( ATTACHMENT_POINTER ) ;
      ParseCONSTRUCT_( CONSTRUCT_POINTER )
end

end ;
```

Here a conditional is parsed. Note the error message creation.

```
PARSE CONDITIONAL_(ATTACHMENT_POINTER)

------------P-  16A  -P-----------------------------------------

       P  1    ...This parses multiple CONDITIONAL_'s by
                   recursive calls from
       P  2    ...Enter ACTUAL_STRING.
   16B P  3    Create CONDITIONAL_(ATTACHMENT_POINTER)
    5D P  4    if REST_OF_LINE blank
    5A P  5        Set ACTUAL_STRING_POINTER to
                      Allocate string(*** No CONDITIONAL FOUND)
```

```
   7C P  6          Accept REST_OF_LINE
      P  7          Locate PARSE_TREE
      P  8          Increment ERROR_COUNT
      P  9       else
   16C P 10          Enter ACTUAL_STRING
      P 11       endif
```

Note how the optional else is handled.

```
PARSE ELSEIF_(ATTACHMENT_POINTER)

------------P-  17A  -P-----------------------------------------

      P  1    ...Optional CONSTRUCT_.
    6A P  2       if Symbol(ELSEIF)
   17B P  3          Create ELSEIF_(ATTACHMENT_POINTER)
    7D P  4          Accept symbol
   16A P  5          Parse CONDITIONAL_(CONDITIONAL_POINTER)
   14B P  6          Parse CONSTRUCT_(CONSTRUCT_POINTER)
   17A P  7          Parse ELSEIF_(ELSEIF_POINTER)
      P  8       endif
```

This is a fully recursive routine.

```
PARSE GROUP_(ATTACHMENT_POINTER)

------------P-  35A  -P-----------------------------------------

      P  1    ...Attempt to find as many GROUP_'s as possible.
    5C P  2       Remove blank lines
    6A P  3       if Symbol(%G)
    7D P  4          Accept symbol
   12C P  5          Create GROUP_(ATTACHMENT_POINTER)
    5D P  6          if REST_OF_LINE blank
    5A P  7             Set GROUP_TITLE_POINTER to
                           Allocate string(*** Missing GROUP TITLE)
    7C P  8             Accept REST_OF_LINE
       P  9             Locate PARSE_TREE
       P 10             Increment ERROR_COUNT
       P 11          else
   10A P 12             Transfer REST_OF_LINE to(GROUP_TITLE)
       P 13          endif
   13A P 14          Parse SEGMENT_(SEGMENT_POINTER)
   35A P 15          Parse GROUP_(GROUP_POINTER)
       P 16       else
       P 17         ...no GROUP_ found - return from Parse GROUP_
       P 18       endif
```

7.3.5. Generate Procedure Segment Tree

The procedures related to this tree provide a group by group, segment by segment walk while creating links of a lexically ordered tree.

The insert and balance algorithm is based on an algorithm by Wirth (1976). A node is initially matched or created. If the creation results in one path becoming more than two units longer than other paths, the tree is reconstructed. This results in a final tree with all paths within two nodes of length. This inplies a future search of order log 2 (N), where N is the number of procedures defined, whilst retaining lexical ordering.

The design definition is given below for the search and insert procedure (this is a complex set of routines).

```
FIND POSITION FOR NEW_PROCEDURE_SEGMENT
     (ATTACHMENT_POINTER,HEIGHT_CHANGE)

-------------P- 38C -F-----------------------------------------

        P  1    ...This algorithm is based on the balanced binary
                        tree algorithms
        P  2    ...proposed by N Wirth in "Algorithms + Data
                        Structures = Programs".
        P  3    ...A full treatment is given in this
                        book. Modifications are made
        P  4    ...in terms of the node structure.
        P  5    ...Binary search for position - note: time is
                        order log(n).
        P  6      if ATTACHMENT_POINTER is NULL
        P  7        ...not in PROCEDURE_SEGMENT_TREE yet
   39A  P  8          Insert NEW_PROCEDURE_SEGMENT
        P  9      else
        P 10          Locate PROCEDURE_SEGMENT at ATTACHMENT_POINTER
        P 11          if NEW_PROCEDURE_NAME left of PROCEDURE_NAME
   39B  P 12            Insert NEW_PROCEDURE_SEGMENT on left
        P 13          elseif NEW_PROCEDURE_NAME right
                            of PROCEDURE_NAME
   41A  P 14            Insert NEW_PROCEDURE_SEGMENT on right
        P 15          else
        P 16            ...already in tree
   42C  P 17              Process NEW_PROCEDURE_SEGMENT
                            is ALREADY_IN_TREE
        P 18          endif
        P 19      endif
```

7.3.6. Generate Explicitly Defined Data Tree

A walk is performed, as for the procedures, adding all explicitly defined data items to the tree. This is performed as above.

7.3.7. Cross Reference

A full walk is performed, first attempting a longest string match to procedures, then to already defined data it[illegible] new data items or non-match tokens. These matches are performed on all [illegible]tionals, procedure lines and general strings at each token boundary.

Each new data item definition is added.

Cross reference linking is done concurrently, adding WHERE_USED and PROCEDURE_USED nodes as required.

This routine is time-critical for the procedures as multiple maximal string matches are performed. Look-up in the trees is optimised by the use of string trees.

The main routines are given below.

```
CROSS REFERENCE

------------P-  51B  -P-----------------------------------------

       P  1    ...Check for invocations of
                        PROCEDURE_SEGMENT's, implicit
       P  2    ...DEFINITION 's and DATA_ITEM usage.
       P  3    ...Start search with each GROUP_.
       P  4     Locate PARSE_TREE
   51C P  5     Cross reference GROUP_(GROUP_POINTER)
```

This is the procedure that cross references the segments within a group, and then the next group recursively. Note the typical tree walk / list processing, recursive structure with a termination check.

```
CROSS REFERENCE GROUP_(ATTACHMENT_POINTER)

------------P-  51C  -P-----------------------------------------

       P  1    ...Search each SEGMENT_.
       P  2     if ATTACHMENT_POINTER is NULL_
       P  3       ...return - last GROUP_ cross referenced
       P  4     else
       P  5        Locate GROUP_ at ATTACHMENT_POINTER
   52A P  6        Cross reference SEGMENT_(SEGMENT_POINTER)
   51C P  7        Cross reference GROUP_(GROUP_POINTER)
       P  8     endif
```

Note the similarity to the above procedure.

```
CROSS REFERENCE SEGMENT_(ATTACHMENT_POINTER)

-------------P-  52A  -P-----------------------------------------

       P  1    ...Search each PROCEDURE_SEGMENT.
       P  2      if ATTACHMENT_POINTER is NULL
       P  3        ...return - last SEGMENT_ in GROUP_
                     cross referenced
       P  4      else
       P  5          Locate SEGMENT_ at ATTACHMENT_POINTER
       P  6          if PROCEDURE_SEGMENT_TYPE
  52B  P  7              Cross reference PROCEDURE_SEGMENT
       P  8          endif
  52A  P  9          Cross reference SEGMENT_(SEGMENT_POINTER)
       P 10      endif
```

This is the first procedure actually to check text, rather than to walk the tree. The procedure line is checked.

```
CROSS REFERENCE PROCEDURE_SEGMENT

-------------P-  52B  -P-----------------------------------------

       P  1    ...Cross references may occur in the
                     PROCEDURE_NAME line or
       P  2    ...in the actual CONSTRUCT's.
       P  3      Locate PROCEDURE_SEGMENT at ATTACHMENT_POINTER
       P  4      Set CURRENT_SEGMENT_POINTER to ATTACHMENT_POINTER
  52C  P  5      Cross reference PROCEDURE_NAME line
  56G  P  6      Cross reference CONSTRUCT_(CONSTRUCT_POINTER)
```

All constructs referenced. Note that as the tree is always corrected, even if the source is faulty, defaults may be assumed.

```
CROSS REFERENCE CONSTRUCT_(ATTACHMENT_POINTER)

------------P-  56C  -P-----------------------------------------

      P  1    ...Search through all CONSTRUCT_'s type by type.
      P  2      if ATTACHMENT_POINTER is NULL_
      P  3        ...return - last CONSTRUCT_ checked
      P  4      else
      P  5          Locate CONSTRUCT_ at ATTACHMENT_POINTER
      P  6          if COMMENT_CONSTRUCT_TYPE
 56D  P  7              Cross reference COMMENT_CONSTRUCT
      P  8          elseif IF_CONSTRUCT_TYPE
 57A  P  9              Cross reference IF_CONSTRUCT
      P 10          elseif WHILE_CONSTRUCT_TYPE
 57D  P 11              Cross reference WHILE_CONSTRUCT
      P 12          elseif UNTIL_CONSTRUCT_TYPE
 58A  P 13              Cross reference UNTIL_CONSTRUCT
      P 14          elseif BLOCK_CONSTRUCT_TYPE
 58B  P 15              Cross reference BLOCK_CONSTRUCT
      P 16          elseif GENERAL_STRING_CONSTRUCT_TYPE
 58C  P 17              Cross reference GENERAL_STRING_CONSTRUCT
      P 18          else
      P 19            ...can only be CASE_CONSTRUCT_TYPE
 58D  P 20              Cross reference CASE_CONSTRUCT
      P 21          endif
      P 22      endif
```

Again just walking *the subservient parts.*

```
CROSS REFERENCE IF_CONSTRUCT

------------P-  57A  -P-----------------------------------------

      P  1    ...Search each subpart.
      P  2      Locate IF_CONSTRUCT at ATTACHMENT_POINTER
 57B  P  3      Cross reference CONDITIONAL_(CONDITIONAL_POINTER)
 56C  P  4      Cross reference CONSTRUCT_(THEN_CONSTRUCT_POINTER)
 57C  P  5      Cross reference ELSEIF(ELSEIF_POINTER)
 56C  P  6      Cross reference CONSTRUCT_(ELSE_CONSTRUCT_POINTER)
 56C  P  7      Cross reference CONSTRUCT_(CONSTRUCT_POINTER)
```

Actually checking text.

```
CROSS REFERENCE CONDITIONAL_(ATTACHMENT_POINTER)

------------P-  57B  -P-----------------------------------------

      P  1    ...Only the string can have references.
      P  2     if ATTACHMENT_POINTER is NULL
      P  3       ...return - last conditional checked
      F  4     else
      P  5         Locate CONDITIONAL  at ATTACHMENT_POINTER
      P  6         Set CURRENT_LINE_NUMBER_POINTER to
                      address of LINE_NUMBER
  59B P  7         Cross reference string
                      (ACTUAL_STRING_POINTER,
                      LINE_NUMBER_POINTER,PROCEDURE_USED_POINTER)
  51B P  8         Cross reference CONDITIONAL
                      (CONDITIONAL_POINTER)
      P  9     endif
```

7.3.8. Generate Line and Page Numbers

Up to this pass, cross references were *made* to logical *page* and line numbers by use of links to data areas. These are now filled-in with the actual numbers using an ordered walk.

7.3.9. Generate Intermediate File

This consists of a number of tree walks. The walks can be in order of appearance (table of contents and *the text itself*), *lexical order* (procedure and data cross references), or in heirarchical order (calling tree).

The main procedure is given below.

```
PREPARE LISTING

------------P-  71B  -P-----------------------------------------

      P  1    ...Emits data for an intermediate file read from
      P  2    ...the PARSE_TREE
  71C P  3     Prepare FRONT_PAGE
  72E P  4     Prepare TABLE_OF_CONTENTS
  74C P  5     Prepare TEXT
  83A P  6     Prepare CALLING_TREE
  86C P  7     Prepare PROCEDURE_INDEX
  87C P  8     Prepare DATA_INDEX
  88B P  9     Prepare LAST_PAGE
      P 10     Close PREPARED_FILE
```

7.3.10. Prepared Data File

The data file is generated by an ordered walk of the full parse tree in the order of information on table of contents, text, calling trees, procedure and data index, and last page. Additional information is provided to allow pagination and indentation.

The lines are sequentially written to a file.

The prepared data structure written to the intermediate file is a simple line-by-line format. For example each "IF" line is an entity on its own, and is sent separately from its matching "ENDIF". Each line is identified by a suitable prefix not shown at this level of detail.

```
1B       1   PREPARED_FILE
2A       2 . FRONT_PAGE
2B       3 . . TITLE_
2E       6 . . REVISION_
3A       8 . . TIME_
3D      11 . . DATE_
4B      15 . TABLE_OF_CONTENTS
4G      16 . . TEXT_PART
4D      17 . . . CONTENT_GROUP
5D      22 . . . CONTENT_PROCEDURE_SEGMENT
6E      30 . . . CONTENT_DATA_DEFINITION_SEGMENT
7B      34 . . . CONTENT_LITERAL_SEGMENT_
7E      38 . . CALLING_TREE_START_INDICATOR
7F      39 . . PROCEDURE_INDEX_START_INDICATOR
8A      40 . . DATA_INDEX_START_INDICATOR
8B      41 . . LAST_PAGE_START_INDICATOR
8C      42 . TEXT_
9A      43 . . NEW_PAGE
9C      46 . . GROUP_
9E      50 . . PROCEDURE_SEGMENT
10A     56 . . DATA_DEFINITION_SEGMENT
10C     60 . . LITERAL_SEGMENT_
10E     64 . . COMMENT_
11C     68 . . IF_
11E     70 . . ELSE_
12B     74 . . ENDIF_
12E     78 . . CONDITIONAL_
13B     82 . . NO_PROCEDURE_USED
13D     84 . . PROCEDURE_USED
13F     87 . . ELSEIF_
14B     89 . . WHILE_
14D     91 . . ENDWHILE_
15A     95 . . REPEAT_
15D     99 . . UNTIL_
15F    101 . . BEGIN_
16C    105 . . END_
16F    109 . . GENERAL_STRING
17B    113 . . CASE_
17D    115 . . ENDCASE_
18A    119 . . OF_
```

```
18C    121  .  .  DEFAULT
18F    125  .  .  DATA_ITEM_ENTRY
19C    130  .  .  COMMENT_ENTRY
19E    134  .  .  LITERAL
20B    138  .  CALLING_TREE
20D    140  .  .  TREE_ENTRY
21B    146  .  .  REPEATED_TREE_ENTRY
21D    153  .  PROCEDURE_INDEX
22A    155  .  .  PROCEDURE_INDEXED
22C    159  .  .  SEGMENT_USED
23C    168  .  DATA_INDEX
23E    170  .  .  DEFINITION_INDEXED
19B    172  .  .  .  DATA_ITEM_NAME (   128 )
24A    173  .  LAST_PAGE
```

7.3.11. Generate Listing

This program does not need the parse tree, and uses only the intermediate file.

It produces a suitable formatted listing from the data, in accordance with basic pagination and indentation rules. The format is defined by use of specific examples for physical format, and an output syntax for logical format. The listing is written to a file, for subsequent editing, display or printing, written to the screen for display, or is printed directly.

A simplified calling tree representation is given below. The main parts of the typical listing are clearly visible. It must be remembered that most parts can be repeated, or are optional.

It can be seen that the intermediate file consists of very small items, each of which can easily be incorporated into the listing.

In the *GENDEC.PAS file, shared by the parser and the formatter programs,* a definition of the identifiers for each of the items is given.

Start of the design for generating the listing - defining the files.

GENCON FILE

```
------------C-   2A  -C-----------------------------------------

       C  1
       C  2      This part of the design is in file GENCON.PDL.
       C  3
       C  4      The other parts of the design are in GENTEXT.PDL
       C  5      and GENIND.PDL.
       C  6
       C  7      This program reads the intermedite file
                   prepared by the
       C  8      parser, and formats it.
       C  9
       C 10      This design should be read in conjunction
                   with the designs
       C 11      the parser, the tree definition, and the
                   prepared file.
       C 12                                                  e
```

The top level design of the listing generator.

GENERATE LISTING

```
------------P-   2B  -P-----------------------------------------

       P  6      Reset PREPARED_FILE
       P  7      Clear OUTPUT_FILE
     3A P  8     Generate FRONT_PAGE
     4C P  9     Generate TABLE_OF_CONTENTS
    10B P 10     Generate TEXT
    22B P 11     Generate CALLING_TREE
    24A P 12     Generate PROCEDURE_INDEX
    26B P 13     Generate DATA_INDEX
    28A P 14     Generate LAST_PAGE
       P 15      Close PREPARED_FILE
       P 16      Close OUTPUT_FILE
```

This is the more detailed calling tree for the listing generation.

```
 2B     1  GENERATE_LISTING
 3A     2  . GENERATE FRONT_PAGE
 4C     9  . GENERATE TABLE_OF_CONTENTS
 10B   26  . GENERATE TEXT
 10C   27  . . GENERATE TEXT_PAGE
 11A   28  . . . GENERATE GROUP_PAGE
 11B   32  . . . GENERATE SEGMENT_PAGE
 11C   35  . . . . GENERATE TEXT_PART
 12A   36  . . . . . GENERATE PROCEDURE_SEGMENT
 12B   37  . . . . . . GENERATE HEADING
                           FOR PROCEDURE_SEGMENT
 13A   38  . . . . . . GENERATE PROCEDURE_PART
 13B   39  . . . . . . . GENERATE COMMENT_OF
                           PROCEDURE_SEGMENT
 14A   40  . . . . . . . GENERATE IF_
 14B   41  . . . . . . . GENERATE ELSE
 14C   42  . . . . . . . GENERATE ENDIF
 15A   43  . . . . . . . GENERATE CONDITIONAL_
 15B   44  . . . . . . . GENERATE NO_PROCEDURE_USED
 16A   45  . . . . . . . GENERATE PROCEDURE_USED
 16B   46  . . . . . . . GENERATE ALSHIP_
 16C   47  . . . . . . . GENERATE WHILE_
 16D   48  . . . . . . . GENERATE ENDWHILE_
 16E   49  . . . . . . . GENERATE REPEAT_
 17A   50  . . . . . . . GENERATE UNTIL_
 17B   51  . . . . . . . GENERATE BEGIN_
 17C   52  . . . . . . . GENERATE END_
 17D   53  . . . . . . . GENERATE GENERAL_STRING
 18A   54  . . . . . . . GENERATE CASE_
 18B   55  . . . . . . . GENERATE ENDCASE_
 18C   56  . . . . . . . GENERATE OF_
 18D   57  . . . . . . . GENERATE DEFAULT
 19A   58  . . . . . GENERATE DATA_DEFINITION_SEGMENT
 20B   62  . . . . . GENERATE LITERAL_SEGMENT
 22B   64  . GENERATE CALLING_TREE
 24A   71  . GENERATE PROCEDURE_INDEX
 26B   80  . GENERATE DATA_INDEX
 28A   88  . GENERATE LAST_PAGE
```

7.4. *Results Obtained*

7.4.1. Use of PDL for the Design

A Program Design Language was used manually for the design of the data structures and the two main programs. The design and the data structures were rewritten into the input format for the processor, and once the programs were written, were processed by the processor.

The sizes of the designs were :

- parse tree design
  - procedure count : 71
  - procedure line count : 298
  - data item count : 136
  - data item usage count : 172

- intermed. d file design
  - procedure count : 133
  - procedure line count : 274
  - data item count : 164
  - data item usage count : 178

- parser design
  - procedure count : 260
  - procedure line count : 2185
  - data item count : 314
  - data item usage count : 2038

- formatter design
  - procedure count : 78
  - procedure line count : 755
  - data item count : 184
  - data item usage count : 576

First doing the design in a program design language, even manually, was beneficial in providing a medium of expressing the concepts, and then being able to evaluate and expand them.

However using the PDL processor on the design input proved to be vastly superior:

- input effort was much less
- formatting was automatic
- updating and changing was easy
- the procedure cross referencing made it easy to go between higher and lower level definitions
- data and procedure indexes provided a very useful checking tool
- the calling tree assisted in providing a quick overview
- the table of contents allowed ready referencing in entry order
- the printed form provided a professional output at little effort

It was felt that using the PDL processor for the design of complex data structures by defining the structures as procedures was very successful. The original intention was not to make use of the processor for this function.

Only four logic errors were found in the entire design since the coding was started. Of these only one was serious - the method of comparing multiple words in text with the defined multiple word procedures - but it is unlikely that this error would have been found if another design methodology had been used. The other errors were trivial.

That only four errors were found to date for a design with 11616 lines of code must be an indication of the merit of the design approach.

7.4.2. Use of Turbo Pascal

The processor was originally intended to be implemented in C on a PDP 11. After the design had already been completed, it was decided to implement the processor on an IBM PC compatible. Turbo Pascal was selected as the language.

No design problems were encountered in the change.

Some time was spent in determining the optimum implementation method for the recursive calls and pointer value transfers and updates, the dynamic data allocation techniques, and the debugging approach.

The data allocation presented some problems - variant records were allocated dynamically - this aspect was not well described in the system documentation, but was finally solved using the GetMem and SizeOf routines.

Debugging support was non-existent, and emphasised that code development and debugging should not be attempted without a symbolic debugger. Methods were evolved of printing the name of each procedure invoked, and routines to print out the trees with their physical link addresses. As any error in a link processing program would cause a complete crash - typically by overwriting the system - finding serious errors was difficult. However because the design and the code was very thoroughly desk checked few errors were found. The most serious errors were in recursive routines where the passed parameter - the link - was not declared as an address variable. This resulted in the system crashing, but only in the next tree walk.

A very good public domain symbolic debugger was located after the bulk of the testing was complete. This would have eased the task immensely.

The integrated editor, compiler and run time environment was good. However the development of a large program - 8000 lines - was too close to the acceptable limit for a floppy disc based system which requires recompilation for every error found.

As the input for the PDL design had first been entered on the PC, the initial steps in creating the code was systematic editting of the source to get it into a Pascal form. This was aided considerably by the use of Superkey - a keyboard macro processor.

Later the software was used on a hard disc system. This reduced compilation times from over eight minutes to :

parser	7929 lines	1:22 minutes
formatter	2915 lines	0:19 minutes
user interface	785 lines	0:04 minutes

The time required to compile is clearly a higher than linear function of the program's size.

The use of Lightning - a disc cacheing program - was particularly useful on a floppy based system, where it reduced disc accesses, and thus disc accessing times by typically 50%.

Code size and efficiency appeared to be fully acceptable - the size was in line with what was originally estimated. This is especially good for a single pass, high speed compiler.

7.4.3. Program Performance

The performance of the processor is limited by the speed of a printer. The processing of the main design, and writing it to disc took 7:54 minutes. Printing would take more than an hour on a typical dot matrix printer (120 cps). About a third of the processing time would be overlapped with the printing. Normal designs would be much smaller than this design - i.e. partitioned into smaller parts. The execution times would be very much shorter as a result. Thus the performance of the processor is believed to be fully acceptable.

The processing time of the parser design to disc was 7:54 minutes, requiring

288 disc reads and 499 disc writes.

The formatter design took just 2:10 minutes.

Some design size statistics :

parser design was	87682 bytes
the intermediate file	332647 bytes
the listing file	449723 bytes

All timing done on an XT compatible in turbo mode 8MHz, with a hard disc, without disc cacheing (disc cacheing tried briefly, but appears to have very minimal effect on time in a hard disc drive system, but reduces disc accesses considerably, which would have an effect on drive life).

Lightning was used to gather disc access information.

7.4.4. Programmer Efficiency

No records were kept of programmer efficiency.

The reasons for this were that all design and coding was done part-time, after work and in leisure hours. This resulted in very wide swings in productivity, and also implied inefficient use of time, with a re-establishing the state of the previous work. Also the designer had reasonable experience in a program design laguage environment, but using assembler and PL/M. This was the first use of Pascal, and of a PC.

Subjectively the efficiency during the period of a year part time during the design, coding and testing phases (i.e. excluding the background search and documenting phases) was what would have been expected of a competant programmer working full time for the same duration. It is clear from this that the efficiency was higher than the more conventional methods of development.

7.4.5. Software Used

MS-DOS version 3.21 by Microsoft.

Turbo Pascal version 3.01 by Borland International.

Turbo Source Code Debugger version 1.02 by L. David Baldwin.

SuperKey version 1.01a by Borland International.

Lightning version 4.16 by Personal Computer Support Group.

8. CONCLUSION

8.1. The Role of Program Design Languages

A Program Design Language is an important tool. It fulfils a role of communication, particularly to the designer himself as well as to others. As a result the design has a high potential of visibility, which if exploited, allows clear logical programs to be constructed.

The simple syntax of Program Design Languages, and the clear design documentation that can be produced using it, allows fellow designers and customers to readily examine, discuss and propose changes to the logical structure of software systems. This builds confidence in the team and the customer that the system will fulfil both the explicit and implicit requirements of the system.

Program Design Languages are compatible with a number of software engineering methodologies. They are relevant irrespective of target computer, language and operating system. As a result, Program Design Languages provide a wide base for software re-use (Walker 1988).

A Program Design Language allows early informal description of algorithms and data. As a result progress is seen early. It also allows design to be detailed to any level, thus ensuring that all logical operations can be explicitly defined.

A Program Design Language illustrates the functional decomposition of a system, and the algorithms and the data structures used. It is supported in this by the calling relationship tree, and procedure and data cross references.

A Program Design Language has a symbiotic relationship with other tools, particularly Data Flow Diagrams.

Data Flow Diagrams define system organisation by displaying external entities, processes, data stores and data flows. Their strength lies with specifying interfaces and data coupling. Their weak points - details of algorithms and data structure - are complemented by the strong points of Program Design Languages.

Additional design information is still required. Major items are the explanation of operation of libraries and packages such as multi-tasking operating systems, and design decision history. These aspects may be covered by written support documentation.

CASE tools have been gaining prominance in the last few years. CASE (Computer Aided Software Engineering) tools have, as their main thrust, been associated with the specification and analysis phases of software. This is an area that had previously been poorly supported by methodologies and tools. Prominent companies have been Cadre Technologies, Cortex, Digital Equipment Corporation, Index Technologies, Promod Incorporated, and Tektronics. Emphasis has been on information modelling, structured analysis and design, design management, configuration control, code generation and tracing, and automatic code generation. It is seen that program design languages fit within the scope of Computer Aided Software Engineering tools.

The establishment of an integrated programmer's workbench where the specification, analysis, design, implementation, and management issues are addressed in a coordinated fashion appears to be a goal that may be reached soon.

In the mean time, program design languages perform a methodology independent role of providing help in the detail design of programs and data.

As far as the future is concerned, Program Design Languages are likely to evolve to interactive syntax-directed editors, with interfaces to provide programming languages support. Some forms of direct symbolic execution is also likely. Whilst this can positively affect design productivity, there are some dangers: - the environment may become target system specific; a tendency may arise to design at the terminal without adequate prior thought; uncontrolled changes may readily be made.

8.2. Formatter Design Conclusions

A formatter program for a Program Design Language is presented.

The technique of bootstrapping was used to create this program. A formatter was designed manually, in a simplified form of the formatter's output. The equivalent input form for the formatter was also produced. These two forms were the results of the first phase. The design was then manually translated to Pascal, and debugged in the second phase of the bootstrap process.

Now the third phase of the bootstrap was reached - the input form of the formatter was processed by the formatter to produce its own design documentation.

An examination of the manually produced simplified form of the design, and the final form indicates a high level of consistency. The additional information provided by the calling relationship tree and the cross references would have substantially eased the design.

The early and complete specification of the formatter was a major factor in the success of the project. This took the form of a user's manual, and detailed input and output formats. The input and output formats were defined using context-free grammars, supplemented by typical samples. This was augmented by rigorous definition, in context-free grammar form, of all expected erroneous inputs. This produced a framework for error reporting and recovery actions.

The design approach used was to exploit the well researched techniques in compiler construction. The method selected was that of recursive descent.

The major achievements were:

- a detailed study of modern design methodologies particularly relating to the detail design phase

- a concise specification of the system via a user manual, which required no significant change during the project

- a complete design before coding which had only four logic errors, with only one being serious

- this error would probably not have been found using any other design methodology

- simple translation into Pascal - which the author had not used previously - instead of C which was originally specified

- no further problems found after initial debugging

- establishing the value of very rigorous design specifications, formal data specifications, design walkthroughs, and the use of a good high level language with debugging support

-the construction of a highly recursive program matching the highly recursive nature of the program design language

- the generation of a totally top-down design including lexical analysis (usually performed bottom-up)

- the generation of a single tree to hold the design, augmented by simple tree walks to include symbol tables, all logical links, line and page numbering, and crossreferences

- the creation of a useful tool

All the design objectives were achieved.

A tool has been created to support software engineering methodologies. This tool can be used as a springboard to produce enhanced forms of design tools. It is also a viable tool, as it stands, for the production of major software systems.

9. REFERENCES

1. Adams, K A; Halasz, I M "25 Ways to Improve Software User Manuals" IEEE Proceedings of Conference on Software Development Tools, Techniques and Alternatives, July 1983, pp 3 - 8.

2. Agarvala, T "Putting Petri Nets to Work" Computer, Volume 7 No 12, December 1979, pp 85 - 94, reprinted in Freeman, P; Wasserman, A I "Tutorial on *Software Design Techniques*" *IEEE Computer Society Press, 4th Edition*, 1984, pp 595 - 604.

3. Aho, A V; Hopcroft, J E; Ullman, J D "The Design and Analysis of Computer Algorithms" Addison-Wesley, 1974.

4. Aho, A V; Sethi, R; Ullman, J D "Compilers Principles, Techniques, and Tools" Addison-Wesley, 1986.

5. Aho, A V; Ullman, J D "Principles of Compiler Design" Addison-Wesley, 1979.

6. Asimov, N "Introduction to Design" Prentice-Hall, 1962.

7. Azuma, M; Tabata, T; Oki, Y; Kamiya, S "SPD : A Humanized Documentation Technology" IEEE Transactions on Software Engineering, Volume SE-11 No 9, September 1985, pp 945 - 953.

8. Backhouse, R C "Syntax of Programming Languages : Theory and Practice" Prentice-Hall, London, 1979.

9. Baker, T "Structured Software Development" INFOTECH State of the Art *Tutorial, Copenhagen, October 1977.*

10. Bailey, R W "Human Performance Engineering : A Guide for System Designers" Prentice-Hall, New Jersey, 1982.

11. Barber, M E "Parsley, A New Compiler - Compiler" IEEE Proceedings of Conference on Software Development Tools, Techniques and Alternatives, July 1983, pp 232 - 241.

12. Bassanino A P; Walker A J "A Syntax-directed PDL Generator for Software Systems Design" Transactions of the SAIEE, Volume 77 No 2, 1986, pp 133 - 143.

13. Bauer, F L; Eickel, J "Compiler Construction - An Advanced Course" Springer-Verlag, Heidelberg, 1974.

14. Beizer, B "Software Testing Techniques" Van Nostrand Reinhold, New York.

15. Bentley, J L "Writing Efficient Programs" Prentice-Hall, New Jersey, 1982.

16. Berg, H K; Giloi, W K "The Use of Formal Specification of Software" Springer-Verlag, Heidelberg, 1979.

17. Bergland, G D "A Guided Tour of Program Design Methodologies" Computer,

Volume 14 No 10, October 1981, pp 13 - 37.

18. Bergland, G D; Gordon, R D "Tutorial - Software Design Strategies" IEEE Computer Society, November 1979.

19. Bochmann, G V "Semantic Evaluation from Left to Right" Communications of the ACM, Volume 19 No 2, February 1976, pp 55 - 62.

20. Boehm, B W "Seven Principles of Software Engineering" Journal of Systems and Software, 1983, Volume 3, pp 3 - 24.

21. Boehm, B W "Software Engineering Economics" Prentice-Hall, New Jersey, 1981.

22. Bohm, C; Jacopini, G "Flow Diagrams, Turing Machines and Languages with only two Formation Rules" Communications of the ACM, Volume 9 No 5, May 1966, pp 366 - 71 reprinted in "Classics in Software Engineering" Ed Yourdon E N, New York : Yourdon Press, 1979, pp 13 - 25.

23. Bornat, R "Understanding and Writing Compilers" MacMillan, London, 1979.

24. Bottos, B A; Kintala, C M R "Generation of Syntax-Directed Editors with Text Oriented Features" Extended Abstract, IEEE Proceedings of Conference on Software Development Tools, Techniques and Alternatives, July 1983, pp 256 - 262.

25. Brooks, F P "The Mythical Man-month" Addison Wesley, Phillipines, 1975.

26. Brown, P J "Writing Interactive Compilers and Interpreters" John Wiley and Sons, Chichester, 1979.

27. Brown, P; "Managing Software Development" Datamation, April 1985, pp 133 - 135.

28. Burstin, M; Porscher, Y; Naimon, Y; Rotbard, I "SuperPDL - A Software Design Tool" IEEE Proceedings of Conference on Software Development Tools, Techniques and Alternatives, July 1983, pp 307 - 313.

29. Buxton, J N; Randell, B "Software Engineering Techniques" Nato Scientific Affairs Division, Brussels, 1969.

30. Caine, S H; Gordon, E K "PDL - A Tool for Software Design" Proceedings National Computer Conference, 1975, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 485 - 490.

31. Callendar, E; Hartsough, C; Kleine, H "SDDL : Software Design And Documentation Language" Proceedings of the NBS IEEE ACM, Software Tool Fair, San Diego, March 1981, pp 44 - 48.

32. Cameron, J R "Two Pairs of Examples in the Jackson Approach to System Development" Proceedings of the 15th Hawaii International Conference on System Sciences, January 1982, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp

201 - 210.

33. Cameron, J R "Tutorial - JSP and SSD : The Jackson Approach to Software Development" IEEE Computer Society, November 1983.

34. Chapin, N "Data Accessibility in Structured Programming" AFIPS Conference Proceedings, Volume 47, 1978, pp 597 - 603.

35. Chapin, N "New Format for Flowcharts" Software Practice and Experience, Vol 4, 1974, pp 341-357.

36. Chu, Y "High-Level Debugging By Interactive Direct Execution" IEEE Proceedings of Conference on Software Development Tools, Techniques and Alternatives, July 1983, pp 274 - 281.

37. Cleaveland, J C; Uzgalis, R C "Grammars for Programming Languages" Elsevier North-Holland, New York, 1977.

38. Cohen, J; Silver, R; Auty, D "Evaluating and Improving Recursive Descent Parsers" IEEE Transactions on Software Engineering, Volume SE-5 No 5, September 1979, pp 472 - 481.

39. Coleman, D; Hughes, J W; Powell, M S "*A Method for the Syntax Directed* Design of Multiprograms" IEEE Transactions on Software Engineering, Volume SE-7, March 1981, pp 189 - 196.

40. Conway, M E "*Design of a Separable Transition-Diagram Compiler*" Communications of the ACM, Volume 6 No 7, July 1963, pp 396 - 408.

41. Curtis, B "Measurement and Experimentation in Software Engineering" *Proceedings of the IEEE, Volume 68 No 9, September* 1980, pp 628 - 641.

42. Davis, A M "The Design of a Family of Application-Oriented Requirements Languages" Computer, Volume 15 No 5, May 1982, pp 21 - 28.

43. Davis, C G; Vick, C R "The Software Development System" IEEE Transactions on Software Engineering, January 1977, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 625 - 640.

44. De Marco, T "Structured Analysis and System Specification" Prentice-Hall, New Jersey, 1979.

45. Dijkstra, E W "Go To Statement Considered Harmful" Communication of the ACM, Volume 11 No 3, March 1968, pp 147 - 148.

46. Dijkstra, E W "The Humble Programmer" Communications of the ACM, October 1972, pp 859 - 866.

47. Donovan, J J "Systems Programming" McGraw-Hill, New York, 1972.

48. Fagan, M E "Design and Code Inspections to Reduce Errors in Program Development" IBM Systems Journal, Volume 15 No 3, 1976, pp 219 - 248, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design

201 - 210.

33. Cameron, J R "Tutorial - JSP and SSD : The Jackson Approach to Software Development" IEEE Computer Society, November 1983.

34. Chapin, N "Data Accessibility in Structured Programming" APIPS Conference Proceedings, Volume 47, 1977, pp 597 - 603.

35. Chapin, N "New Format for Flowcharts" Software Practice and Experience, Vol 4, 1974, pp 341-357.

36. Chu, Y "High-Level Debugging By Interactive Direct Execution" IEEE Proceedings of Conference on Software Development Tools, Techniques and Alternatives, July 1983, pp 274 - 281.

37. Cleaveland, J C; Uzgalis, R C "Grammars for Programming Languages" Elsevier North-Holland, New York, 1977.

38. Cohen, J; Silver, R; Auty, D "Evaluating and Improving Recursive Descent Parsers" IEEE Transactions on Software Engineering, Volume SE-5 No 5, September 1979, pp 472 - 481.

39. Coleman, D; Hughes, J W; Powell, M S "A Method for the Syntax Directed Design of Multiprograms" IEEE Transactions on Software Engineering, Volume SE-7, March 19E , pp 189 - 196.

40. Conway, M E "Design of a Separable Transition-Diagram Compiler" Communications of the ACM, Volume 6 No 7, July 1963, pp 396 - 408.

41. Curtis, B "Measurement and Experimentation in Software Engineering" Proceedings of the IEEE, Volume 68 No 9, September 1980, pp 628 - 641.

42. Davis, A M "The Design of a Family of Application-Oriented Requirements Languages" Computer, Volume 15 No 5, May 1982, pp 21 - 28.

43. Davis, C G; Vick, C R "The Software Development System" IEEE Transactions on Software Engineering, January 1977, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 625 - 640.

44. De Marco, T *"Structured Analysis and System Specification"* Prentice-Hall, New Jersey, 1979.

45. Dijkstra, E W "Go To Statement Considered Harmful" Communication of the ACM, Volume 11 No 3, March 1968, pp 147 - 14..

46. Dijkstra, E W "The Humble Programmer" Communications of the ACM, October 1972, pp 859 - 866.

47. Donovan, J J "Systems Programming" McGraw-Hill, New York, 1972.

48. Fagan, M E "Design and Code Inspections to Reduce Errors in Program Development" IBM Systems Journal, Volume 15 No 3, 1976, pp 219 - 248, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design

Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 548 - 576.

49. Farrow, R "Generating a Production Compiler from an Attribute Grammar" IEEE Software, Volume 1 No 4, October 1984, pp 77 - 93.

50. Farrow, R "Linguist - 86 : Yet Another Translator Writing System Based on Attribute Grammars" ACM SIGPLAN Notices, Volume 17 No 6, June 1982.

51. Foster, J M "List Processing" MacDonald and Jane's American Elsevier, London, 1967.

52. Foster, J M "Automatic Syntactic Analysis" MacDonald and Jane's American Elsevier London, 1970.

53. Freeman, P "Fundamentals of Design" Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Press, 4th Edition, 1984, pp 2 - 22.

54. Freeman, P "Reusable Software Engineering : Concepts and Research Directions" Proceedings of the Workshop on Reusability in Programming, September 1983, *reprinted in Freeman, P; Wasserman, A I "Tutorial on Software* Design Techniques" IEEE Computer Press, 4th Edition, 1984, pp 63 - 76.

55. Freeman, P "Towards Improved Review of Software Designs" Proceedings of the National Computer Conference, 1975, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 542 - 547.

56. Freeman, P "Requirements Analysis and Specification : The First Step" ASME, Advances in Computer Technology - 1980, August 1980, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE *Computer Society Press*, 4th Edition, 1984, pp 79 - 85.

57. Gane, C P; Sarson, T "Structured System Analysis : Tools and Techniques" Prentice-Hall, New Jersey, 1979.

58. Gane, C P "Data Design in Structured Systems Analysis" INFOTECH State of the Art Report on Data Design, 1980, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 115 - 132.

59. Gonnet, G H "Handbook of Algorithms and Data Structures" Addison-Wesley, *London*, 1984.

60. Goos, G; Hartmanis, J "Software Engineering - An Advanced Course" Springer-Verlag, Heidelberg, 1973.

61. Graham, R M in Naur, P; Randell, B "Software Engineering" Nato Scientific Affairs Division, Brussels, January 1969, quoted in Yourdon, E "Techniques of Program Structure and Design" Prentice Hall, New Jersey, 1975, p xi.

62. Graham, S L "Table Driven Code Generation" IEEE Computer, Volume 17 No 8, August 1980.

63. Grogono, P "Programming in Pascal" Addison-Wesley, 1980.

64. Halstead, M H "Elements of Software Science" Elsevier, New York, 1977.

65. Hamilton, M; Zeldin, S "The Relationship Between Design and Verification" Journal of Systems and Software, Volume 1, 1979, pp 29 - 56, reprinted in Freeman, P; Wasserman, A I "Tutorial of Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 641 - 668.

66. Hoare, C A R "Communicating Sequential Processes" Communications of the ACM, August 1978, pp 666 - 677, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 366 - 377.

67. Hunt, J W "Tutorial Series - 15 : Programming Languages" Computer, Volume 15 No 4, April 1982, pp 70 - 88.

68. Hurley, R B "Decision Tables in Software Engineering" Van Nostrand Reinhold, New York, 1983.

69. Jackson, M A "Constructive Methods of Program Design" Proceedings of the First Conference of the European Co-operation in Informatics, Volume 44, 1976, pp 236 - 262 reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 514 - 532.

70. Jackson, M A "Principles of Program Design" Academic Press, London, 1975.

71. Jackson, M A "Information Systems : Modeling, Sequencing and Transformations" McKeag, R M; MacNaghten, A M "On the Construction of Programs" Cambridge University Press, Cambridge, 1980, pp 319 - 341.

72. Jackson, M A "System Development" Prentice-Hall, New Jersey, 1983.

73. Jamsa, K; Kameroff, S "Turbo Pascal Programmers Library" Osbourne McGraw-Hill, 1986.

74. Jensen, K; Wirth,N "Pascal User Manual and Report" Second Edition, Springer-Verlag,New York, 1975.

75. Jensen, R W "Tutorial : Structural Programming" Computer, Volume 14 No 3, March 1981, pp 31 - 48.

76. Jensen, R W; Tonies, C C "Software Engineering" Prentice-Hall, New Jersey, 1979.

77. Johnson, S C "YACC : - Yet Another Compiler-Compiler" Computing Science Technical Report 32, Bell Laboratories, Murrey Hill, 1975.

78. Kernighan, B W; Plauger, P J "The Elements of Programming Style" McGraw-Hill, 1974.

79. Kernighan, B W; Plauger, P J "Software Tools" Addison-Wesley Publishing Company, 1976.

80. Kernighan, B W; Ritchie, D M "The C Programming Language" Prentice-Hall, New Jersey, 1978.

81. Knuth, D E "Semantics of Context-Free Languages" Mathematical Systems Theory, Volume 2 No 2, 1968 pp 127 - 145. Addendum in Volume 5 No 1, 1971, pp 95 - 96.

82. Kopetz, H "Software Reliability" MacMillan, 1979.

83. Lauber, R J "Development Support Systems" Computer, Volume 15 No 5, May 1982, pp 36 - 46.

84. Ledgard, H F; Chmura, L J "Fortran With Style" Hayden, New Jersey, 1978.

85. Lesk, M E "Lex - A Lexical Analyser Generator" *Computing Science* Technical Report, 39, Bell Laboratories, Murray Hill, 1975.

86. Lewis, P M; Rosenkrantz, D J; Stearns, R E "Attributed Translations" Journal of Computer and System *Sciences, Volume 9, 1974, pp 29 - 307.*

87. Lewis, P M; Rosenkrantz, D J; Stearns, R E "Compiler Design Theory" Addison-Wesley, 1976.

88. Lewis, P M; Stearns, R E "Syntax-Directed Transduction" Journal of the ACM, Volume 15 No 3, July 1968, pp 465 - 488.

89. Marcotty, M; Ledgard H F "A Sample of Formal Definitions" Computing Surveys, Volume 8 No 2, June 1976, pp 191 - 276.

90. McCabe, T J "A Complexity Measure" IEEE Transactions on Software Engineering, Volume SE-2 No 4, December 1976, pp 308 - 320.

91. McGettrick, A D "The Definition of Programming Languages" Cambridge University Press, 1980.

92. Metzger, P W "Managing a Programming Project" Prentice-Hall, New Jersey, 1973.

93. Miller, G A "The Magical Number Seven, Plus or Minus Two : Some Limits on our Capacity for Processing Information" Psychology Review, Volume 63, pp 81 - 97 March 1956, reprinted in Norman, D A, "Memory and Attention : An Introduction to Human Information Processing" John Wiley and Sons, New York, 1969, pp 73 - 80.

94. Mills, H D "Software Development" IEEE Transactions on Software Engineering, Volume SE-2 No 4, December 1976, pp 265 - 273.

95. Mills, H D "*How to Write Correct Programs and Know It*" *Proceedings* of the International Conference on Reliable Software, ACM SIGPLAN Notices, Volume 10 No 6, 1975, pp 363 - 370.

96. Myers, G J "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections" Communications of the ACM, Volume 21 No 9, September 1978, pp 515 - 524.

97. Myers, G J "Advances in Computer Architecture" John Wiley and Sons, New York, Second Edition, 1982.

98. Myers, G J "Software Reliability" Wiley-Interscience, New York, 1976.

99. Myers, G J "The Art of Software Testing" John Wiley and Sons, New York.

100. Nassi, I; Shneiderman, B "Flowchart Techniques for Structured Programming" SIGPLAN Notices, Volume 8 No 8, 1973.

101. Naur, P et al "*Report on the Algorithmic Language* ALGOL 60" Communications of the ACM, May 1960, pp 299 - 314.

102. Naur, P; Randell, B "Software Engineering" Nato Scientific Affairs Division, *Brussels, January 1969.*

103. Noonan, R E "Structured Programming and Formal Specification" IEEE Transactions on Software Engineering, Volume SE-1 No 4, December 1975, pp 421 - 425.

104. - PDL Program Design Language Reference Guide (Processor Version 3), Warren Point Computers, Hertfordshire, February 1977.

105. - "Process Design Methodology" Volume 1, Process Design Language Specifications, Texas Instruments, Huntsville, A L, December 1975.

106. Parnas, D L "On the Criteria to be used in Decomposing Systems into Modules" Communications of the ACM, Volume 15 No 12, December 1972, pp 1053 - 1058, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 304 - 309.

107. Parnas, D L "Designing Software for Ease of Extension and Contraction" IEEE Transactions on Software Engineering, Volume SE-5, No 2, March 1979, pp 128 - 138, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 310 - 320.

108. Parnas, D L; Clements, P C; Weiss, D M "The Modular Structure of Complex Systems" IEEE Transactions on Software Engineering, Volume SE-11 No 3, *March 1985, pp 259 - 266.*

109. Peters, L J "Software Design : Methods and Techniques" Yourdon Press, New York.

110. Peters, L J "Software Representation and Composition Techniques" Proceedings of the IEEE, Volume 68 No 9, September 1980, pp 1085 - 1093.

111. Peters, L J; Tripp, L L "Software Design Representation Schemes" Proceedings of the Symposium on Computer Software Engineering, Polytechnic

Press, New York, 1976.

112. Peterson, W W; Kasami, T; Tokura, N "On the Capabilities of While, Repeat and Exit Statements" Communications of the ACM, August 1973, pp 503 - 512.

113. Pozefsky, D; Jazayeri, M "A Family of Pass-Oriented Attribute Grammar Evaluators" Proceedings of the ACM, 1978 Conference, pp 261 - 270.

114. Privitera J P "ADA Design Language for the Structural Design Methodology" *Proceedings ADATEC Conference*, October 1982, pp 76 - 90, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 491 - 505.

115. Ramanathan, J; Blattner, M "Program Forms and *Program Form Analysers* for High-Level Structured Design" AFIPS Conference Proceedings, Volume 48, 1979, pp 1013 - 1020.

116. Riddle, W E; Fairley, R E "Software Development Tools" Springer-Verlag, Heidelberg, 1980.

117. Rohl, J S "*An Introduction to Compiler Writing*" MacDonald and Jane's American Elsevier, London, 1975.

118. Rohl, J S "Recursion via Pascal" Cambridge University Press, Cambridge, 1984.

119. Rosenkrantz, D J; Stearns, R E "Properties of Deterministic Top-Down Grammars" Information and Control, Volume 17, 1970, pp 226 - 256.

120. Ross, D T; Goodenough, J B; Irvine, C A "Software Engineering : Process, Principles, and Goals" Computer, Volume 8 No 5, May 1975, pp 17 - 27.

121. Ross, D T; "Applications and Extensions of SADT" Computer, Volume 18 No 4, April 1985, pp 25 - 34.

122. Ross, D T "*Structured Analysis (S A) : A Language for Communicating Ideas*" IEEE Transactions on Software Engineering, Volume SE-3 No 1, January 1977, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Press, 4th Edition, 1984, pp 96 - 114.

123. Rubin, L F "Syntax-Directed Pretty Printing - A First Step To A Syntax-Directed Editor" IEEE Transactions on Software Engineering, Volume SE-9 No 2, March 1983, pp 119 - 127.

124. Shaw, M "Abstraction Techniques in Modern Programming Languages" IEEE Software, Volume 1 No 4, October 1984.

125. Shooman, M L "Software Engineering" McGraw-Hill, 1983.

126. Sommerville, I "Software Engineering" Addison-Wesley, 1982.

127. Stanovic, J A "Software Communication Mechanisms : Procedure Calls Versus Messages" Computer, Volume 15 No 4, April 1982, pp 19 - 25.

128. Stevens, W P; Myers, G J; Constantine. L L "Structured Design" IBM Systems Journal, Volume 13 No 2, 1974 pp 115 - 139, reprinted in Freeman, P; Wasserman, A I Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 328 - 352.

129. Sutton, S A; Basili, V R "The Flex Software Design System : Designers Need Languages, Too" Computer, Volume 14 No 11, November 1981, pp 95 - 102.

130. Tennant, R D "Principles o Programming Languages" Prentice-Hall,New York, 1981.

131. Van Leer, P "Top-Down Development using a Program Design Language" I B M Systems Journal, Volume 15 No 2, 1976.

132. Waite, W M; Goos, G "Compiler Construction" Springer-Verlag, Heidelberg, 1984.

133. Walker, A J "Structured Information Processing System Design" Revision 1.1, February 1984, Revision 2.0, November 1984, Revision 4.0, June 1986, Course Material, Department of Electrical Engineering, University of the Witwatersrand, Johannesburg.

134. Walker, A J "A software challenge for the 90's" Electron, May 1988, pp 9-13.

135. Wasserman, A I "Information System Design Methodology" Journal of the American Society for Information Science, January 1980, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Press, 4th Edition, 1984, pp 43 - 62.

136. Wasserman, A I "The User Software Engineering Methodology : An Overview" in Olle,T W; Sol, H G; Verrijn-Stuart, A A "Information Systems Design Methodologies: a Comparitive Review", 1982, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Press, 4th Edition, 1984, pp 669 - 713.

137. Wasserman, A I "Tutorial : Programming Language Design" IEEE Computer Society, October 1980.

138. Weinberg, G M; Schulman, E L "Goals and Performance in Computer Programming" Human Factors, Volume 16 No 1, 1974, pp 70 - 77.

139. Weinberg, G M "The Psychology of Computer Programming" Van Nostrand Reinhold, New York, 1971.

140. Weiner, R; Sincovec, R "Software Engineering with Modula 2 and Ada" Johan Wiley and Sons, New York, 1984.

141. Weinwurm, G F "On The Management of Computer Programming" Auerbach, New York, 1970.

142. Welsh, J "A Structured Compiler" McKeag, R M; MacNaghten, A M in "On the Construction of Programs" Cambridge University Press, Cambridge, 1980, pp

59 - 105.

143. Welsh, J; McKeag, M "Structured System Programming" Prentice-Hall, London, 1980.

144. Wetherall, C; Shannon, A "Tidy Drawing of Trees" IEEE Transactions on Software Engineering, Volume SE-5 No 5, September 1979, pp 514 - 520.

145. Wirth, N "Program Development by Stepwise Refinement" Communications of the ACM, April 1971, pp 221 - 227, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 533 - 539.

146. Wirth, N "On The Composition of Well-Structured Programs" Computing Surveys, Volume 6, No 4, Decemb [illegible] 4, pp 247 - 259.

147. Wirth, N "Algorithms + Data Structures = Programs" Prentice-Hall, New Jersey, 1976.

148. Woodson, T T "Introduction to Engineering Design" McGraw-Hill, New York, 1966.

149. Yoder, C M; Schrag, M L "Nassi-Shneiderman Charts : An Alternative to Flowcharts for Design" Proceedings of the ACM SIGSOFT/SIGMETRICS Software Quality and Assurance Workshop, November 1978, pp 306 - 393, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 506 - 513.

150. Young, S "P-Notation : High Level Notation Description Language for Software Design" Microprocessors and Microsystems, Volume 4 No's 7 to 10, 1980.

151. Yourdon, E "Techniques of Program Structure and Design" Prentice-Hall, New Jersey, 1975.

152. Yourdon, E "Design of On-Line Computer Systems" Prentice-Hall, New Jersey, 1972.

153. Yourdon, E "Managing the Structured Techniques" Second Edition, Prentice-Hall, New Jersey, 1979.

154. Yourdon, E "Structured Walkthroughs" Yourdon Press, New York, 1978.

155. Zelkowitz, M V; Shaw, A C; Gannon, J D "Principles of Software Engineering and Design" Prentice-Hall, New Jersey, 1979.

156. Zelkowitz, M V; Yeh, R T; Hamlet, R G; Gannon, J D; Basili, V R "Software Engineering Practices in the U S and Japan" Computer, Volume 17 No 6, June 1984, pp 57 - 66.

A USER MANUAL FOR
A PROGRAM DESIGN LANGUAGE PROCESSOR

Andreas Martin Dinkelspiel

Version 0 Revision 3

JOHANNESBURG, 1988

Table of Contents

1. INTRODUCTION

A program design language is a tool for software development. Such a language has a similar role to design tools like flowcharts, structure charts and HIPO.

It is a simple, useful way of presenting a design from an early stage. The design can then be read, reviewed and refined. The detailed design can readily be translated into code, and remain as useful documentation for debugging, testing, training and maintenance.

The language allows the program and its data to be clearly presented in a readable format. The entry of the design is simple. Finally, useful references and indexes are automatically provided by the formatter.

It is assumed that the user of this manual has some knowledge of a programming language, the editor, and sufficient knowledge of the operating system to run programs and list files.

2. SAMPLE SESSION

This chapter provides a practical overview of software design, the role of a program design language, and how the language is used. A simple problem is defined. A solution is generated and illustrated.

2.1. Problem

A program is required to check materials to build roof trusses. The three lengths of the triangular truss are entered. This program must state whether the truss is scalene (no lengths equal), isosceles (two lengths equal), or equilateral (all lengths equal).

2.2. Approach

The problem is first analysed for completeness and correctness. Further detail may be provided: a terminal is available to enter and display the result; the program will be initiated as part of another program; only one analysis is required.

An initial simple solution is generated. The solution consists of partitions of the whole problem into sub-problems. It is then examined at that level. If not satisfactory, another solution may be generated. Once an acceptable solution is found at that level, the components are considered. This is the phase in which the program design language is used.

Once the design has been detailed to a level where no design decisions remain, it can be translated into code, debugged, tested and used.

2.3. Using the Program Design Language

The program design language, and its formatter, are illustrated using a simple example. The example is taken through several stages, resulting in a detailed design. The formatter outputs are discussed for this example, and some explanation is given for the methods used.

2.3.1. First Stage

The first level defined may be:-

Enter Lengths; Check Validity; Display Type

This is already a viable design. It may be typed into a file. The procedure can be given a name e.g. "Truss Type Analyser".

The entry is:-

```
XP Truss Type Analyser
Enter Lengths
Check Validity
Display Type
```

The design can now be formatted. The output is a full listing including all indexes, tables of contents etc.

Once satisfied with this level, the design can be detailed by expanding the single procedure, or by providing lower level procedures. We have provided the further procedures:-

```
XP Enter Lengths
Print "Type in Lengths of Truss"
Set FIRST_LENGTH to Number Read
Set SECOND_LENGTH to Number Read
Set THIRD_LENGTH to Number Read

XP Check Validity
Check Lengths are greater than zero
Check that Lengths form a Triangle

XP Display Type
IF FIRST_LENGTH is equal to SECOND_LENGTH
IF FIRST_LENGTH is equal to THIRD_LENGTH
Print "Equilateral Triangle"
ELSE
Print "Isosceles Triangle"
ENDIF
ELSEIF FIRST_LENGTH is equal to THIRD_LENGTH
Print "Isosceles Triangle"
ELSEIF SECOND_LENGTH is equal to THIRD_LENGTH
Print "Isosceles Triangle"
ELSE
Print "Scalene Triangle"
ENDIF
```

We have now defined data items (FIRST_LENGTH, SECOND_LENGTH and THIRD_LENGTH) by forming one or more words connected by underscores. A single word followed by an underscore is also regarded as a data item.

We have also defined three additional procedures, the last of which has nested selection constructs.

After formatting, our Table of Contents is expanded, and we have some additional entries in our Calling Tree, Procedure Index and Data Index.

Our design may again be evaluated and refined. We decide to expand the "Check Validity" procedure's entries by adding:-

```
XP Check Lengths are greater than zero
IF FIRST_LENGTH is greater than zero\
and SECOND_LENGTH is greater than zero\
and THIRD_LENGTH is greater than zero
... All Lengths are greater than zero
ELSE
Print "Lengths must be greater than zero"
ENDIF

XP Check that Lengths form a Triangle
... No triangle exists if one side is longer than the other
... two sides together
IF FIRST_LENGTH is longer than SECOND_LENGTH and THIRD_LENGTH
Print "Does not form a triangle"
ENDIF
IF SECOND_LENGTH is longer than FIRST_                    RD_LENGTH
Print "Does not form a triangle"
ENDIF
IF THIRD_LENGTH is longer than FIRST_LENG                 OND_LENGTH
Print "Does not form a triangle"
ENDIF
```

The use of a comment line and a conditional continuator (a condition on an if, while, until etc., that covers more than one line) is shown in "Check Lengths are greater than zero".

The design can now again be formatted and evaluated. At this stage, the design would be trivial to translate to a computer language such as Basic. However, careful evaluation shows that processing would continue after an error is found.

This example is illustrated in the appendix as PDL2A.PDL.

The design is changed as follows:-

```
XP Truss Type Analyser
Enter Lengths
Check Triangle
IF TRIANGLE_VALID
Display Type
ENDIF

XP Enter Lengths
Print "Type in Lengths of Truss"
Set FIRST_LENGTH to Number Read
Set SECOND_LENGTH to Number Read
Set THIRD_LENGTH to Number Read
```

```
XP Check Triangle
Check Lengths are greater than zero
IF LENGTHS_VALID
Check that Lengths form a Triangle
ELSE
Set TRIANGLE_VALID to FALSE_
ENDIF

XP Check Lengths are greater than zero
IF FIRST_LENGTH is greater than zero\
and SECOND_LENGTH is greater than zero\
and THIRD_LENGTH is greater than zero
... All Lengths greater than zero
Set LENGTHS_VALID to TRUE_
ELSE
Set LENGTHS_VALID to FALSE_
Print "Lengths must be greater than zero"
ENDIF

XP Check that Lengths form a Triangle
... No triangle exists if one side is longer than the other
... two sides together
IF FIRST_LENGTH is longer than SECOND_LENGTH and THIRD_LENGTH
Print "Does not form a triangle"
Set TRIANGLE_VALID to FALSE_
ELSEIF SECOND_LENGTH is longer than FIRST_LENGTH and THIRD_LENGTH
Print "Does not form a triangle"
Set TRIANGLE_VALID to FALSE_
ELSEIF THIRD_LENGTH is longer than FIRST_LENGTH and SECOND_LENGTH
Print "Does not form a triangle"
Set TRIANGLE_VALID to FALSE_
ELSE
... All combinations checked
Set TRIANGLE_VALID to TRUE_
ENDIF

XP Display Type
IF FIRST_LENGTH is equal to SECOND_LENGTH
IF FIRST_LENGTH is equal to THIRD_LENGTH
Print "Equi.ateral Triangle"
ENDIF
ELSEIF FIRST_LENGTH is equal to THIRD_LENGTH
Print "Isosceles Triangle"
ELSEIF SECOND_LENGTH is equal to THIRD_LENGTH
Print "Isosceles Triangle"
ELSE
Print "Scalene Triangle"
ENDIF
```

The full listing is illustrated in the Appendix called PDL2B.PDL.

2.4. Referencing

A full listing is provided in the appendix with all indexes.

The listing includes a table of contents, the groups, procedure, data definition and comment segments entered, the calling hierarchy, and the procedure and data index.

2.4.1. Table of Contents

The table of contents shows the page numbers for the automatically generated sections as well as all the procedures entered.

2.4.2. Procedure Segments

The procedure segments themselves have line numbers for the contents. In addition, where procedures are invoked which are defined elsewhere, a page reference is also given.

The data items have no referencing in the procedures, but are visible because of their underscores.

Indentation is also performed.

2.4.3. Calling Hierarchy

An indented list of procedure segments, and the pages on which they are to be found is produced.

The highest level is on the left-hand side. Each procedure is followed by the procedures it invokes, indented further to the right.

This hierarchy gives a global overview of the structure of the software. It fulfills a similar role to structure charts or HIPO hierarchy diagrams.

2.4.4. Procedure Index

The procedure index lists all defined procedures alphabetically. Each entry indicates the page on which it is defined, and all procedures, in order of appearance, which invoke it (if any). The invoking procedures' page is given, as well as each line.

2.4.5. Data Index

The data index lists all data items alphabetically. Each entry indicates all segments, in order of appearance, which use the data item. The procedure's individual line numbers, which access the data item, are also given.

2.5. Design Refinement

As soon as a preliminary version of the design is formulated, it is entered and listed. As a result of examining the listing, the design is modified or refined to show further levels of detail. This ensures early documentation and a basis for further work.

At times, as further insight into the problem and design is gained, the design may need to be revised. This is inherent in all design processes - if we initially knew the best final solution we would not need to design at all.

The approach of using simple nested procedures allows the complexity of a procedure under consideration to be limited. This allows the design to be simple.

3. DESIGN METHODOLOGY

3.1. Purpose

A design methodology is a collection of methods, techniques and practices. It forms a framework to direct activities. As a result effort is systematically applied to one aspect of the problem at a time.

In addition a framework allows standardisation in approach, design and documentation.

A program design language is an aid to understanding and documenting the logic of designs. It presents the algorithms and data of a design in a simple to read and understand format. The entry is also simplified to minimise effort.

Finally, a program design language formatter can provide some automated support for indexing and cross-referencing. Error checking of the design is simplified. Changes to be made can easily be correlated to the existing design to indicate affected procedures and data structures.

3.2. Scope

Once any feature of a program is known, it can be defined and documented by a program design language. Typical features include comments, written descriptions, procedures and data structures.

As additional insight into the problem is gained, the design can be updated.

Thus a very high level design can be detailed until the implementation in a computer language is straight-forward. During this process the design can be checked and modified.

During implementation the design is translated into the computer language. Minor design amendments can be made to correspond to languages, operating system and input/output device dependencies.

During debugging and testing the design provides the logic whereas the code provides the implementation details. The debugging and testing consists of verifying that the code does correspond to the design, and separately, that the design fulfills user needs.

Implementation errors are corrected in the code only, while logic errors are corrected in the design first, and then the code is re-written to reflect the changes.

The design is retained to document the program, and to facilitate maintenance and upgrading.

The scope of a program design language thus extends from specification and conceptual design, to maintenance.

3.3. Results

Program design languages have gained the highest acceptance of all design

tools, They have been used since the mid-70's, and are thus well tested. Ease of use, clear designs and lov numbers of errors in the final code are the main advantage of program design languages.

4. LISTING FORMAT

The listing produced by the formatter has the following sections:-

- Front Page
- Table of Contents
- Procedure, Comment and Data Definition Segments
- Calling Trees
- Procedure Index
- Data Index

4.1. Front Page

The front page consists of a design title, its revision, and time and date of processing.

Input

The title is defined by a line starting with XTITLE. The contents of the remainder of the line are reproduced on the front page. Only one title is allowed, but it is optional. This is the first line of the design input. If omitted, a default of "DESIGN LISTING" is reproduced. The first 32 characters of the line are also reproduced on top of every page.

The revision line starts with XREVISION. The contents of the remainder of the line are reproduced on the front page under the title. Only one revision line is allowed, but is optional. This is the second line of the design input. If omitted, a default of "CURRENT REVISION" is reproduced.

The time and date are automatically provided by the formatter. The date is also reproduced on top of each subsequent page.

4.2. Table of Contents

This is automatically generated by the formatter. The groups, and comment, data definition, and procedure segments defined by the design input are listed. The groups are at the left of the pages, while the procedure, comment and data definition segments are indented one level to the right.

Page numbers are given on the right-hand side.

4.3. Groups

It is often useful to partition a design into parts. These may be separate tasks, procedures, or data sections. The parts are defined by group lines in the input. For each group line, a page is reproduced containing the design title and the group line. The group line is also reproduced on the second line of subsequent procedure or data pages, until overwritten by the next group line. The group page is also indexed in the table of contents.

Group lines are optional, and if omitted, the second line of procedure and data pages is left blank.

Input

The group is defined by a line starting with XG.

4.4. Procedure Segments

Procedure segments are used to define the procedural or functional parts of programs. A procedure may be empty (i.e. only its name defined), or may have one or more lines. The format of these lines is defined in a later section. A maximum number of lines is defined to correspond to one page.

As many segments of the same group as possible are reproduced on each page. This allows optimum use of paper.

Input

A procedure is defined by a line starting with %P. It is terminated by the subsequent procedure, data definition, group or end of input.

The rest of the line is reproduced on the procedure name line.

In addition, the name part and the parameter part only of the line is reproduced as required, in the table of contents, calling hierarchy, procedure index and data index.

4.5. Data Definition Segments

Data items may be defined within procedures by simply entering a sequence of words joined by single underscores. A single word followed by a single underscore is also recognised as a data item. This is called implicit definition.

Data items may be defined in data definition segments. This is the preferred method of defining more complex data items, or those used by several procedures. The detailed data item syntax is defined in a later section.

Multiple data definitions may be reproduced on one page.

Input

A data definition segment is defined by a line starting with %D.

It is terminated by a subsequent procedure, comment segment, data definition segment, group or end of input.

The rest of the line is reproduced on the data definition name line.

In addition, the rest of the line is reproduced as required, in the table of contents and data index.

4.6. Comment Segments

A comment segment is defined by a line starting with %C. It is terminated by the subsequent procedure, data definition segment, group, comment segment or end of input.

The rest of the line is reproduced on the comment segment name line.

In addition, the rest of the line is reproduced in the table of contents.

All subsequent lines in the comment segment are reproduced, totally unchanged, with a line number on the left. They may thus be used for descriptions, tables, block diagrams etc.

4.7. Calling Trees

These are generated automatically from the procedures defined, and calls to these procedures.

In general, a procedure is given in the tree, followed by the procedures it calls indented towards the right. Each of these procedures in turn are followed by the procedures they call further indented to the right.

Two special cases exist:-

- where one set of procedures is invoked from more than one place in the design (common calling trees)

- where a procedure invokes itself, either directly or via intermediate procedures (recursion)

Common calling trees are not repeated for each procedure invoking it. Instead, the first invocation would have the tree, and the others only the first common procedure. A reference is provided to indicate where the common tree is defined.

In the case of recursion, no tree exists for calling relationships, but rather cyclic graphs. A tree is still provided, but at the first point where the tree is repeated an indication is given and the search for relationship is stopped.

A separate tree is created for each set of procedures. The lower level procedures may be common to more than one tree.

Multiple trees are presented on one page. A tree may also span more than one page.

For each entry in the calling tree, the relevant page number is given.

4.8. Procedure Index

An alphabetically arranged index of all defined procedures is generated automatically, with their page numbers.

For each procedure there is given, in order of appearance, each procedure, page and line number where it is referenced.

4.9. Data Index

An alphabetically arranged index of all defined data items is generated automatically. Two types of entries exist:-

- explicitly defined data items (from data definition segments)

- implicitly defined data items

Explicitly defined data items have entries, in order of appearance, of data definition segment, page, and line number where they are defined, as well as each procedure, page and line number where they are used.

Implicitly defined data items only have entries for each procedure, page and line number where they are used in order of appearance.

5. SYNTAX

A program design language, like any language, is composed of characters, words, sentences, paragraphs and sets of paragraphs.

Unlike natural languages, a program design language has a relatively small set of valid constructs with a small set of forming rules, or syntax.

In this chapter an introduction to the principal forms for procedures and data definitions is given.

5.1. Procedure Syntax

A procedure consists of the first line and the optional procedure body. The first line defines its name, and is explained in the section on Procedure Segments in the previous chapter.

The procedure body syntax allows the definition of the design logic. The parts of the procedure body are called constructs.

The types of constructs are:-

- comment construct
- if construct
- while construct
- until construct
- case construct
- procedure construct
- general string construct

Any number of these may be provided in sequence. However, no procedure should extend to over one page. There are two reasons for this:-

- indentation levels would not be visible
- the procedure would be unnecessarily complex

Whenever a procedure becomes too long, it should be converted to two or more simpler procedures.

5.1.1. Comment Construct

A comment is used to provide a single line of additional information which is not directly part of the logic. It can also be used to visually separate parts of a procedure.

Input

A comment is defined by three consecutive full stops on the left-hand side of a line. It may contain any other characters in the rest of the line. The contents are not *analysed. The line is reproduced indented but unchanged.*

Example

```
... This is a comment line
... So is this and the next line
...
```

5.1.2. If Construct

The if construct is used whenever part of the logic is optional dependent on *conditions.*

The if construct has the following parts:-

- if line
- then part
- elseif parts
- else part
- endif line

It defines the construct, and provides the condition for the then logic to be used. The line is started by an "if". The conditional line can include one or more procedures, data items and operators such as OR, AND, +, = etc.

If the conditional line contains procedures, this line will be referenced in the procedure index.

If the conditional line includes data items, these will also be referenced by the data index.

If a single conditional line is not enough to express the design, the current line is terminated with a "\". This allows the next line to be an extension of this line.

The then part *consists of zero, one, or more constructs.*

Zero, one, or more elseif parts may follow. Each elseif part consists of the elseif line, followed by zero, one, or more constructs.

The elseif line is started by an "elseif", followed by a conditional line, with the same form as that in the "if" line. The elseif parts may be used to select one of a number of alternatives.

The else part is also optional, but only one may be used for each construct. It consists of a single line which may only have the word "else" on it, *followed by zero, one, or more constructs.*

The endif part is the final part. One and only one must be provided. It consists of a single line which may only have the word "endif" on it.

Input

An if construct is recognised by starting with an if line, and is terminated with an endif line. It may include elseif lines and an else line. The contents between the if line, elseif lines, else line and endif line are indented one further position to the right.

Example

```
if NUMBER_ is one
print "you have chosen an apple"
elseif NUMBER_ is two
print "you have chosen an orange"
else
print "you may only enter a one or a two"
endif
```

Note that the if construct may nest other constructs, including itself:

```
if the animal has fur
if the animal has a pouch
print "it may be a kangaroo"
endif
endif
```

5.1.3. While Construct

This construct may be used whenever a part of the logic may be performed not at all, once, or many times.

The construct has three parts:-

- *the while line*

- zero, one, or more constructs

- the endwhile line

The while line is always required. It defines the construct, and provides the condition for the logic to be performed. The line is started by a "while". The conditional line is treated the same as the if construct's conditional line.

The constructs can be any constructs, including while constructs.

The endwhile line is always required. It consists of only an "endwhile" on a line.

Input

The while construct is recognised by a while line, and is terminated by the endwhile line. The contents between these lines are indented one position *further to the right.*

Example

```
while the child is hungry
give it a biscuit
endwhile
```

Note that the use without another construct is also useful.

```
find door
while the door is locked
endwhile
open door
```

This is a "wait" condition.

5.1.4. Until Construct

This construct may be used whenever a part of the logic will always be *performed at least once.*

The construct has three parts:-

- the repeat line

- zero, one, or more constructs

- the until line

The repeat line is always required. It defines the extent of the until construct.

The constructs can be any constructs, including until constructs.

The until line is always required. It consists of the "until" and a conditional line, similar to that of the if construct.

Input

The until construct is defined by the repeat line, and is terminated by the until line. The contents between these lines are indented one position further to the right.

Example

```
repeat
add petrol to the tank
until the tank is full
```

Note that use without another nested construct is also useful:

```
open the valve
repeat
until tank is full
close the valve
```

5.1.5. Case Construct

The case construct is useful, instead of the if construct, when a large number of identical tests, with differing matching conditions must be performed.

The construct has these parts:-

- the case line
- zero, one, or more switches, each with zero, one, or more constructs
- an optional default line, with zero, one, or more constructs
- the endcase line

The case line is always required. It defines the construct, and provides the test for the logic to be performed. The line is started by a case. The conditional line is treated as in the if construct.

The switches define particular matching conditions, and the logic to be performed. A switch consists of the "OF", a list of at least one matching condition, and the applicable constructs, if any.

The matching conditions have the same form as a conditional line.

The default line indicates the logic to be performed when no switch has a matching condition. The default is optional. Only one is allowed. It consists solely of "default" on the line.

The endcase line indicates the end of the case construct. It is always required. It consists of a line with only "endcase" on it.

Input

The case construct is defined by the case and endcase lines. All constructs within the case are indented one further position to the right.

The case construct may contain any other nested constructs, including the case construct.

Example

```
case fruit
of orange, grapefruit
peel
eat
of grape
pick berries
eat
of apple
remove core
eat rest
default
print "only orange, grapefruit, grape or apple valid"
endcase
```

5.1.6. Procedure Construct

This single line construct may be used to invoke separately defined logic into the current procedure. The construct may be considered to be equivalent to the procedure body that it references.

The construct is actually a subset of the general string construct, but is presented separately because of its importance.

The construct consists of:-

- the procedure name

- an optional parameter list

The procedure name is the character string up to the optional parameter list.

A comparison is made with all defined procedure segments to determine a match with this string. If a match is found, this is illustrated by a reference to the defined procedure on the left-hand side.

An optional parameter list may be defined. This consists of a left bracket,

zero, one, or more data items separated by commas, and a right bracket.

The parameter field is not checked against the defined procedure for consistency. This allows different data items to be included. The onus is on the designer to verify parameter consistency.

Input

The input format is simply the name and the parameters (if required). If no match is found to a defined procedure, it is treated as a general string construct. If a match is found, a reference to it is given on the left-hand side of the line.

Example

```
Output ( CARRIAGE_RETURN, LINE_FEED )
... enables feed of nutrients
Open valve
```

5.1.7. General String Construct

This single line construct is used to provide all the general logic. It can be used for statements similar to those in program langueges. Any operators, such as OR, AND, NOT, =, + etc. may be included. It also includes the procedure construct defined above.

A general string construct may be used as a procedure construct for procedures which are as yet undefined. Once they are defined, the line construct becomes a procedure construct.

A null line consists of only a blank line. This is useful to separate parts of the input, or to separate procedures for input. A null line is totally removed in the listing.

The general string construct is examined for data items. These are then referenced in the data index.

Input

The general string construct consists of zero, one, or more characters. These may be any alloved by the system.

Example

```
Y_ = X_ + 1

NEXT_VALUE = sin ( X_ + 0,3 )

Print "this is a line"
```

5.2. Data Definition Segment

Data is a most important part of designs. The data is defined in two ways:-

- explicitly in a data definition segment

- implicitly by just using a data item in procedures

A data definition segment consists of a data segment line prefixed with %D, and an optional data segment body. The data segment line consists of a name of one or more characters.

The optional data segment consists of:-

- data item lines used to define data items

- comment constructs used to provide a single line of information

- blank lines used only to separate inputs, completely removed from listings

Input

A data definition is defined by a %D followed by its name. It is terminated by the next procedure definition, data definition segment, comment segment, group or end of file.

No indentation is performed on its body.

Example

%D Global variables

5.2.1. Data Item Lines

A data item line defines data. It consists of a valid data item and a string of zero, one, or more characters.

The data item consists of words formed from letters or digits separated by underscores. A single word followed by an underscore is also a data item. The longest part conforming to the form of a data item is accepted as its definition. The subsequent string is not interpreted.

Input

The data item line consists of a data item and a possible string. The page of *definition will be included in the data index.*

Example

VALID_RECORD_POINTER used to indicate next valid record

PERSONNEL_STARTDATE

5.2.2. Comment Construct

This is identical to that used in procedure definitions above.

5.2.3. Implicit Data Definition

A data item can be defined in any non-comment part of a procedure. It will then be entered in the data index, but no definition page is given.

5.3. Comment Segment

A comment segment is defined by a line starting with %C. It is terminated by the subsequent procedure, data definition segment, group, comment segment or end of input.

The rest of the line is reproduced on the comment segment name line.

In addition, the rest of the line is reproduced in the table of contents.

All subsequent lines in the comment segment are reproduced, totally unchanged, with a line number on the left. They may thus be used for descriptions, tables, block diagrams etc.

5.4. General Syntax

In order to simplify input entry, additional form feeds, blank lines tabs and spaces may be used. In the segments and constructs where the input is interpreted (procedure and data definition segments), tabs and multiple spaces are interpreted as single spaces. Form feeds are interpreted as blank lines.

Whenever a line exceeds the available width of the paper, the right-hand part is truncated, and a final three "*"'s are given on the line.

6. ADVANCED FEATURES

This program design language is intended to be simple for the occasional user, yet to be powerful enough for the advanced user. Certain features are not normally required except for the advanced user. These are defined below. Also some additional technical explanations are given.

6.1. Parameter Passing

The program design language has support for parameter passing. A procedure may be defined with formal parameters. The onus rests with the designer to check that the number, order and type of parameters match in all cases.

The procedure may itself be a return parameter (ie. it can be used as a function), by defining its name in the form of a data item and using a general string construct starting with "return".

Example

```
%P INVERSE_ ( NUMBER_ )
return 1/NUMBER_
```

Another example is:-

```
%P SUM_ ( FIRST_NUMBER, SECOND_NUMBER )
return FIRST_NUMBER + SECOND_NUMBER
```

6.2. Recursion

Recursion is a technique where a given procedure invokes itself. The invocation may be direct or indirect. In other words, the procedure may invoke itself, or it may invoke other procedures, which in turn invoke the first procedure.

Recursion is a powerful concept for problems which have an inherent recursive nature. Here the use of a recursive design may be elegant and concise. It is a fact however, that wherever a recursive design is used, it may be replaced by a similar design using while or until constructs.

It must also be recognised that the indiscriminate use of recursion may increase the effort required to design, test and understand a design.

The references include works defining the use of recursion.

The formatter supports the use of recursion. It does modify the calling tree entries, however, to prevent an endless sequence. At the first point where a procedure invokes itself, the procedure's name is entered followed by the number of the line on which it was first listed in the tree.

The factorial function for a number N greater than one:-

```
XP FACTORIAL_ ( N_ )
If N_ = 1
return N_
else
return N_ * FACTORIAL_ ( N_ - 1 )
endif
```

This can also be performed using a while construct.

```
XP FACTORIAL_ ( N_ )
RESULT_ = N_
while N_ is greater than 1
N_ = N_ - 1
RESULT_ = RESULT_ * N_
endwhile
return RESULT_
```

6.3. Scope of Names

The scope of all names is global ie. any data item or procedure may be accessed from any part of the design.

If an *implementation language* or a *design* standard has particularly restrictive scope, this may be enforced by the designer. The provision of calling trees, and procedure and data index makes checks for consistent scope usage simple.

A procedure may also be invoked ahead of its definition, or within its definition (recursion).

7. DETAILED SYNTAX

In this chapter a detailed syntax is given. It must be noted that all valid designs must have valid syntax (as valid sentences in English must conform to accepted grammar rules). The use of valid syntax is not sufficient for a valid design, however (as a sentence "The bone eats a dog." has the correct parts but has no valid meaning). A valid design must also have valid semantics, which is not readily defined in a formal way.

7.1. Notation

In order to define the syntax, a notation convention is useful. A suitable method is BNF (Backus Naur Form). This consists of a uniform method of defining final sets of characters as entered in the design, using names for intermediate levels and meta symbols to present the syntax.

The intermediate levels of the syntax are defined as a left parenthesis followed by a simple description, followed by a right parenthesis: (design), (page number) etc.

The meta symbols are left and right parenthesis (as above), the "is re-written as" operator of "::=", the "optional part" shown as left and right square brackets, and the "or" meta symbol "|".

The final sets of characters are shown directly. Where more than one option exists, these are listed one below the other, or separated by the "or" meta symbol "|".

For example:-

```
( types of fruit ) ::= Orange
                       Pear
                       Apple
```

or

```
( types of fruit ) ::= Orange | Pear | Apple
```

This can be read as "'types of fruit' is rewritten as Orange, Pear or Apple".

Optional parts are defined using the square brackets :-

```
( signed number ) ::= [( sign )]( unsigned number )

( sign ) ::= + | -
```

This defines that no "sign" need be used, but if it is used, it must be + or -.

A more complex example is :-

(page number) ::= (non-zero digit)[(digit)][(digit)]

(digit) ::= (non-zero digit)
0

(non-zero digit) ::= 1|2|3|4|5|6|7|8|9

This can be read as "a 'page number' is re-written as a 'non-zero digit' followed by an optional 'digit' followed by an optional 'digit'. A 'digit' is re-written as a 'non-zero digit' or a zero. A 'non-zero digit' is re-written as one or two or three or four or five or six or seven or eight or nine".

In other words, wherever (page number) is found in the syntax it implies that any number from 1 to 999 is syntactically correct, but that 0 or 029 are not valid. Further (semantic) rules are still needed to enforce sequential numbers from 1, without any missing numbers.

7.2. Valid Syntax

Reserved words (all words appearing in the input syntax such as "if", "while", "case", "TITLE' etc.) can be entered in upper or lower case.

```
 1 )   ( design )
        ::=
          [( title )][( revision )][( text block )]
             ( end of file )

 2 )   ( title ) ::= %TITLE ( title line )

 3 )   ( title line )
        ::=
          ( first title string )[( second title string )]
             ( end of line )

 4 )   ( first title string ) ::= ( string )

 5a )  ( string ) ::= ( character )( string )
 b )                  ( character )

 6a )  ( character ) ::= ( digit )
 b )                     ( letter )
 c )                     ( other character )

 7 )   ( digit ) ::= 0|1|2|3|4|5|6|7|8|9

 8a )  ( letter ) ::= ( upper case letter )
 b )                  ( lower case letter )

 9 )   ( upper case letter ) ::= A|B|C|D|E|F|G
                                 H|I|J|K|L|M|N
                                 O|P|Q|R|S|T|U
                                 V|W|X|Y|Z

10 )   ( lower case letter ) ::= a|b|c|d|e|f|g
                                 h|i|j|k|l|m|n
                                 o|p|q|r|s|t|u
                                 v|w|x|y|z

11a )   ( other character ) ::= ( printer dependent )
 b )                             +|-|/|%|:|;|'|.|,|?

12 )   ( second title string ) ::= ( string )

13 )   ( end of line ) ::= ( carriage return, line feed )

14 )   ( revision ) ::= %REVISION ( revision line )
                           ( end of line )

15 )   ( revision line ) ::= ( string )
```

```
16a )   ( text block ) ::= ( text )( text block )
 b )                     | ( text )

17a )   ( text ) ::= ( comment segment )
 b )               | ( procedure segment )
 c )               | ( data segment )
 d )               | ( group )

18 )    ( comment segment ) ::= %C ( literal line )[( literals )]

19 )    ( literal line ) ::= ( literal string )( end of line )

20 )    ( literal string ) ::= ( string )

21a )   ( literals ) ::= ( literal )( literals )
 b )                   | ( literal )

22 )    ( literal ) ::= ( single line string )

23 )    ( single line string ) ::= [( string )]( end of line )

24 )    ( procedure segment )
        ::=
           %P ( procedure line )[( procedure body )]

25 )    ( procedure line )
        ::=
           ( procedure string )[( parameter tail )]
              ( end of line )

26 )    ( procedure string ) ::= ( string )

27 )    ( procedure body ) ::= ( constructs )

28 )    ( parameter tail )
        ::=
           ( left bracket )[( string )]( right bracket )
              [( string )]


29a )   ( constructs ) ::= ( construct )( constructs )
 b )                     | ( construct )

30a )   ( construct ) ::= ( comment construct )
 b )                   | ( if construct )
 c )                   | ( while construct )
 d )                   | ( until construct )
 e )                   | ( case construct )
 f )                   | ( block construct )
 g )                   | ( procedure construct )
 h )                   | ( general string construct )

31 )    ( comment construct ) ::= ...( single line string )
```

```
32 )   ( if construct ) ::= ( if line )( if body )
                             ( endif line )

33 )   ( if line ) ::= IF ( conditional line )

34a )   ( conditional line )
        ::=
           ( string )( end of line )
 b )       ( string )( additional lines )( end of line )

35a )   ( additional lines )
        ::=
           ( additional line )( additional lines )
 b )       ( additional line )

36 )   ( additional line )
        ::=
           ( continuator )( end of line )[( string )]

37 )   ( if body ) ::= [( then part )][( elseif parts )]
                       [( else part )]

38 )   ( then part ) ::= ( constructs )

39a )   ( elseif parts ) ::= ( elseif part )( elseif parts )
 b )                        ( elseif part )

40 )   ( elseif part ) ::= ( elseif line )[( constructs )]

41 )   ( elseif line ) ::= ELSEIF ( conditional line )

42 )   ( else part ) ::= ( else line )[( constructs )]

43 )   ( else line ) ::= ELSE ( end of line )

44 )   ( endif line ) ::= ENDIF ( end of line )

45 )   ( while construct )
        ::=
           ( while line )[( constructs )]( endwhile line )

46 )   ( while line ) ::= WHILE ( conditional line )

47 )   ( endwhile line ) ::= ENDWHILE ( end of line )

48 )   ( until construct )
        ::=
           ( repeat line )[( constructs )]( until line )

49 )   ( repeat line ) ::= REPEAT ( end of line )

50 )   ( until line ) ::= UNTIL ( conditional line )
```

51) (case construct)
::=
(case line)[(switches)][(default)]
(endcase line)

52) (case line) ::= CASE (conditional line)

53a) (switches) ::= (switch)(switches)
b) (switch)

54) (switch) ::= OF (conditional line)[(constructs)]

55) (default) ::= DEFAULT (end of line)[(constructs)]

56) (endcase line) ::= ENDCASE (end of line)

57) (block construct)
::=
(begin line)[(constructs)](end line)

58) (begin line) ::= BEGIN (end of line)

59) (end line) ::= END (end of line)

60) (procedure construct) ::= (procedure line)

61) (general string construct) ::= (single line string)

62) (data segment)
::=
%D (data segment line)[(data segment body)]

63) (data segment line)
::=
(data definition string)(end of line)

64) (data definition string) ::= (string)

65) (data segment body) ::= (definitions)

66a) (definitions) ::= (definition)(definitions)
b) (definition)

67a) (definition) ::= (data item line)
b) (comment construct)

68) (data item line) ::= (data item)[(string)]
(end of line)

69a) (data item) ::= (word)_(word tail)
b) (word)_

70a) (word) ::= (word part)(word)

b) (word part)

71a) (word part) ::= ([illegible])

b) (d[illegible])

72a) (word tail) ::= (word)(word tail)

b) (word)

73) (group) ::= ZG (group line)

74) (group line) ::= (group string)(end of line)

75) (group string) ::= (string)

7.3. Complete Syntax

The complete syntax is defined to consider all possible syntax entries, valid or invalid. The syntax is presented, with error rules indicated by an asterisk.

```
 1 )   ( design ) ::= [( title )][( revision )]
                      [( text block )]( end of file )

2a )   ( title ) ::= %TITLE ( title line )
 b )                 %TITLE ( end of line ) *

 3 )   ( title line )
       ::=
          ( first title string )[( second title string )]
             ( end of line )

 4 )   ( first title string ) ::= ( string )

5a )   ( string ) ::= ( character )( string )
 b )                  ( character )

6a )   ( character ) ::= ( digit )
 b )                     ( letter )
 c )                     ( other character )
 d )                     ( invalid character ) *

 7 )   ( digit ) ::= 0|1|2|3|4|5|6|7|8|9

8a )   ( letter ) ::= ( upper case letter )
 b )                  ( lower case letter )

 9 )   ( upper case letter ) ::= A|B|C|D|E|F|G
                                 H|I|J|K|L|M|N
                                 O|P|Q|R|S|T|U
                                 V|W|X|Y|Z

10 )   ( lower case letter ) ::= a|b|c|d|e|f|g
                                 h|i|j|k|l|m|n
                                 o|p|q|r|s|t|u
                                 v|w|x|y|z

11a )   ( other character ) ::= ( printer dependent )
 b )                            +|-|/|%|:|;|.|,|?

12 )   ( second title string ) ::= ( string )

13 )   ( end of line ) ::= ( carriage return, line feed )

14a )   ( revision ) ::= %REVISION ( revision line )
                            ( end of line )
 b )                    %REVISION ( end of line ) *
```

15) (revision line) ::= (string)

16a) (text block) ::= (text)(text block)
b) (text)

17a) (text) ::= (comment segment)
b) (procedure segment)
c) (data segment)
d) (group)

18a) (comment segment)
::=
%C (literal line)[(literals)]
b) % (literal line)[(literals)] *
c) (literal line)[(literals)] *

19a) (literal line) ::= (literal string)(end of line)
b) (end of line) *

20) (literal string) ::= (string)

21a) (literals) ::= (literal)(literals)
b) (literal)

22) (literal) ::= (single line string)

23) (single line string) ::= [(string)](end of line)

24a) (procedure segment)
::=
%P (procedure line)[(procedure body)]
b) %P (end of line)[(procedure body)] *

25a) (procedure line)
::=
(procedure string)[(parameter tail)]
(end of line)
b) (parameter tail)(end of line) *

26) (procedure string) ::= (string)

27) (procedure body) ::= (constructs)

28a) (parameter tail)
::=
(left bracket)[(string)](right bracket)
[(string)]
b) (left bracket)[(string)] *

29a) (constructs) ::= (construct)(constructs)
b) (construct)

30a) (construct) ::= (comment construct)
b) (if construct)

```
 c )                        ( while construct )
 d )                        ( until construct )
 e )                        ( case construct )
 f )                        ( block construct )
 g )                        ( procedure construct )
 h )                        ( general string construct )

31 )    ( comment construct ) ::= ...( single line string )

32a )    ( if construct ) ::= ( if line )( if body )
                                 ( endif line )
 b )                          ( if line )( if body ) *

33 )    ( if line ) ::= IF ( conditional line )

34a )    ( conditional line )
         ::=
            ( string )( end of line )
 b )        ( string )( additional lines )( end of line )
 c )        ( end of line ) *

35a )    ( additional lines )
         ::=
            ( additional line )( additional lines )
 b )        ( additional line )

36 )    ( additional line )
         ::=
            ( continuator )( end of line )[( string )]

37 )    ( if body ) ::= [( then part )][( elseif parts )]
                        [( else part )]

38 )    ( then part ) ::= ( constructs )

39a )    ( elseif parts ) ::= ( elseif part )( elseif parts )
 b )                         ( elseif part )

40 )    ( elseif part ) ::= ( elseif line )[( constructs )]

41 )    ( elseif line ) ::= ELSEIF ( conditional line )

42 )    ( else part ) ::= ( else line )[( constructs )]

43a )    ( else line ) ::= ELSE ( end of line )
 b )                       ELSE ( string )( end of line ) *

44a )    ( endif line ) ::= ENDIF ( end of line )
 b )                        ENDIF ( string )( end of line ) *

45a )    ( while construct )
         ::=
            ( while line )[( constructs )]( endwhile line )
 b )        ( while line )[( constructs )] *
```

```
46 )   ( while line ) ::= WHILE ( conditional line )

47a )  ( endwhile line ) ::= ENDWHILE ( end of line )
 b )                       ENDWHILE ( string )( end of line ) *

48a )  ( until construct )
       ::=
          ( repeat line )[( constructs )]( until line )
 b )      ( repeat line )[( constructs )] *

49a )  ( repeat line ) ::= REPEAT ( end of line )
 b )                       REPEAT ( string )( end of line ) *

50 )   ( until line ) ::= UNTIL ( conditional line )

51a )  ( case construct )
       ::=
          ( case line )[( switches )][( default )]
             ( endcase line )
 b )      ( case line )[( switches )][( default )] *

52 )   ( case line ) ::= CASE ( conditional line )

53a )  ( switches ) ::= ( switch )( switches )
 b )                    ( switch )

54 )   ( switch ) ::= OF ( conditional line )[( constructs )]

55a )  ( default )
       ::=
          DEFAULT ( end of line )[( constructs )]
 b )      DEFAULT ( string )( end of line )[( constructs )] *

56a )  ( endcase line ) ::= ENDCASE ( end of line )
 b )                        ENDCASE ( string )( end of line ) *

57a )  ( block construct )
       ::=
          ( begin line )[( constructs )]( end line )
 b )      ( begin line )[( constructs )] *

58a )  ( begin line ) ::= BEGIN ( end of line )
 b )                      BEGIN ( string )( end of line ) *

59a )  ( end line ) ::= END ( end of line )
 b )                    END ( string )( end of lin. ) *

60 )   ( procedure construct ) ::= ( procedure line )

61 )   ( general string construct ) ::= ( single line string )

62 )   ( data segment )
       ::=
```

```
                %D ( data segment line )[( data segment body )]

63a )   ( data segment line )
        ::=
          ( data definition string )( end of line )
 b )        ( end of line ) *

64 )    ( data definition string ) ::= ( string )

65 )    ( data segment body ) ::= ( definitions )

66a )   ( definitions ) ::= ( definition )( definitions )
 b )                        ( definition )

67a )   ( definition ) ::= ( data item line )
 b )                       ( comment construct )
 c )                       ( invalid definition )

68 )    ( data item line ) ::= ( data item )[( string )]
                                ( end of line )

69a )   ( data item ) ::= ( word )_( word tail )
 b )                      ( word )_

70a )   ( word ) ::= ( word part )( word )
 b )                 ( word part )

71a )   ( word part ) ::= ( letter )
 b )                      ( digit )

72a )   ( word tail ) ::= ( word )_( word tail )
 b )                      ( word )

73 )    ( invalid definition ) ::= ( string )( end of line ) *

74 )    ( group ) ::= %G ( group line )

75a )   ( group line ) ::= ( group string )( end of line )
 b )                       ( end of line ) *

76 )    ( group string ) ::= ( string )
```

8. ERROR MESSAGES

Errors are recognised at the appropriate points in the operation of the program, and self-explanatory messages are inserted in the formatted output. A minimum amount of error correction is attempted. The last page provides an error count.

The program used performs standard error checks using MSDOS and Turbo Pascal, in which it is written. These consist mainly of checking file names, paths and attributes, and space availability on the discs.

The following processing errors are identified during the formatting process :

*** No CONDITIONAL found

*** String must not be on ELSE line

*** String must not be on ENDIF line

*** Missing ENDIF

*** String must not be on ENDWHILE line

*** Missing ENDWHILE

*** Missing UNTIL

*** String must not be on REPEAT line

*** String must not be on DEFAULT line

*** String must not be on ENDCASE line

*** Missing ENDCASE

*** String must not be on BEGIN line

*** String must not be on END line

*** Missing END

*** Missing PROCEDURE NAME before parenthesis

*** Missing RIGHT PARENTHESIS

*** Missing PROCEDURE SEGMENT NAME

*** DEFINITION must start with a DATA ITEM or "..."

*** Missing DATA DEFINITION NAME

*** MISSING LITERAL SEGMENT NAME

*** Missing P, D, C, or G

*** Missing %G, %P, %D, or %C

*** Missing GROUP TITLE

*** Duplicated PROCEDURE NAME

*** DATA ITEM already defined

8.1. Relation to Complete Syntax

The use of incorrect syntax will result in an error message. In order to demonstrate what the incorrect syntax usage was, a reference is given to a detailed syntax indicating the error detected.

The combined correct and incorrect syntax is given in the previous chapter.

8.2. Error Checklist

Certain types of errors occur frequently in practice. A good method of checking designs is to systematically check the design from beginning to end for each type of error.

The calling tree, procedure and data index are particularly useful for checking certain types of errors.

A checklist is given below:-

- all data items initialised before use
- data items accessed where expected
- data items not accessed where not expected
- procedures used where expected
- no procedures used where not expected
- all data items accessed at least twice
- each procedure invoked indicates a reference (did the names match)
- any aliases for data items (eg. RECORD_NUMBER, RECORD_NO, NO_OF_RECORD)
- the calling tree reflects your understanding of the design structure
- recursion only occurs where expected
- any "orphans" in the calling tree (procedures defined but not used)
- any comment segments where procedures were expected

- where data definition segments are used, the data index has entries with no definition reference

9. OPERATION

9.1. Installation

The operation of the formatter program is defined in this chapter. The programs are supplied on a floppy disc. The programs are written to execute on an IBM PC or compatible using MSDOS 2.11 or higher.

The formatter consists of three ".COM" files : PDL.COM, INITA.COM, and GENIND.COM. PDL.COM checks that the other two .COM files are present, then requests information for the formatting, INITA.COM analyses the input PDL, preparing an intermediate file, which is then formatted using the GENIND.COM file.

These files are expected to be in the default path.

In addition to these files, the intermediate file (PDLPRE.PDL) generated by the formatter must be taken into account. This file contains the full design information in an unformatted form, and is approximately two thirds the size of the full formatted form. This file is automatically generated and updated as required.

9.2. Processing

Initially the source is prepared using any editor that produces normal ASCII format files. Typical editors are EDLIN, Wordstar in the "non-document" form, and the editor of Turbo Pascal.

Processing is then performed by invoking the PDL.COM file.

The PDL program will then request input and output filenames, workspace path, lines per page and columns per page, and then cause execution of the INITA.COM and GENIND.COM files.
The input filename request has a default path which is the same with which the formatter was invoked. Any other path can be given, however. The suffix, or extension, of the file is expected to be .PDL, but any other is accepted. If the filename is not valid (wrong path or missing file), the user can exit the program by typing a carriage return.

The output filename request has a default of the printer (type a carriage return). Otherwise the current path is again assumed, as well as a suffix of .LST. Again any other suffix can be given.

A standard approach can be to use the same filename for the input and output files, with suffixes of .PDL and .LST.

For a screen preview, enter an output filename of CON:, and control scrolling with Control-S.

The workspace path on which the intermediate file PDLPRE.PDL is written is next requested. The default is on the current path. If insufficient space exists on this drive, any other path may be given. This may be useful if the source is on a floppy, and the intermediate file is put on a RAM disc, or on a hard disc.

The lines per page used for the output is requested, with a default of 72, and a range of 50 to 90.

The columns per page available for the output is requested. The default is 120, and a range of 79 to 132.

The formatter now processes the design. During its operation, the user is given an indication of progress by the display of the states:

Reading in text from file - note disc access.
Checking text for all procedure segments.
Scanning the text for the explicitly defined data items.
Cross-referencing all procedure invocations and data usages.
Line and page numbers added.
Preparing the intermediate file - note disc accesses.
Printing front page and table of contents.
Printing the actual text.
Printing the calling tree.
Printing the procedure index.
Printing the data index.
Completed processing of the design.

Note that at any stage execution may be terminated by typing Control-C. The intermediate files will then be deleted by the next execution of the program.

10. REFERENCES

Some general references are given below.

Classical papers on program design languages are by Van Leer and Caine and Gordon.

A detailed survey of program design are given by Kernigan and Plauger (two references), Welsh and McKeag, and Zelkowitz, Shav and Gannon.

Good programming primers are given by Grogono, Kernigan and Ritchie, and Wirth.

1. Caine, S H; Gordon, E K "PDL - A Tool for Software Design" Proceedings National Computer Conference, 1975, reprinted in Freeman, P; Wasserman, A I "Tutorial on Software Design Techniques" IEEE Computer Society Press, 4th Edition, 1984, pp 485 - 490

2. De Marco, T "Structured Analysis and System Specification" Prentice Hall, New Jersey, 1979

3. Gane, C P; Sarson, T "Structured System Analysis : Tools and Techniques" Prentice-Hall, New Jersey, 1979

4. Grogono, P "Programming in Pascal" Addison-Wesley, 1980

5. Jackson, M A "Principles of Program Design" Academic Press, London, 1975

6. Jensen, R W; Tonies, C C "Software Engineering", Prentice-Hall, New-Jersey, 1979

7. Kernigan, B W; Plauger, P J "The Elements of Programming Style" McGraw-Hill, 1974

8. Kernigan, B W; Plauger, P J "Software Tools" Addison-Wesley, 1976

9. Kernigan, B W; Ritchie, D M "The C Programming Language" Prentice-Hall, New Jersey, 1978

10. Ledgard, H F; Chmura, L J "Fortran with Style" Hayden, New Jersey, 1978

11. Peters, L J "Software Design : Methods and Techniques" Yourdon Press, New York, 1981

12. Van Leer, P "Top-Down Development Using a Program Design Language" I B M Systems Journal, Volume 15 No 2, 1976

13. Walker, A J *"Structured Information Processing Systems Design" Revision* 4.0 June 1986, Course Material, Department of Electrical Engineering, University of the Witwatersrand, Johannesburg

14. Welsh, J; McKeag, M "Structured Systems Programming" Prentice-Hall, London, 1980

15. Wirth, N "Algorithms + Data Structures = Programs" Prentice-Hall, New Jersey, 1976

16. Yourdon, E "Techniques of Program Structure and Design" Prentice-Hall, New Jersey, 1975

17. Zelkowitz, M V; Shaw, A C; Gannon, J D "Principles of Software Engineering and Design" Prentice-Hall, New Jersey, 1979

****

DESIGN LISTING

CURRENT REVISION

Time 17 : 34

Date 30 7 1988

****

Truss Type Analyser

```
-------------P-  1A  -P----------------------------------------

    1B P  1     Enter Lengths
    1C P  2     Check Validity
    2A P  3     Display Type
```

Enter Lengths

```
-------------P-  1B  -P----------------------------------------

       P  1     Print "Type in Lengths of Truss"
       P  2     Set FIRST_LENGTH to Number Read
       P  3     Set SECOND_LENGTH to Number Read
       P  4     Set THIRD_LENGTH to Number Read
```

Check Validity

```
-------------P-  1C  -P----------------------------------------

    2B P  1     Check Lengths are greater than zero
    3A P  2     Check that Lengths form a Triangle
```

Display Type

```
----------P-  2A  -P-----------------------------------------------
          P  1     If FIRST_LENGTH is equal to SECOND_LENGTH
          P  2        If FIRST_LENGTH is equal to THIRD_LENGTH
          P  3           Print "Equilateral Triangle"
          P  4        else
          P  5           Print "Isosceles Triangle"
          P  6        endif
          P  7     elseIf FIRST_LENGTH is equal to THIRD_LENGTH
          P  8        Print "Isosceles Triangle"
          P  9     elseIf SECOND_LENGTH is equal to THIRD_LENGTH
          P 10        Print "Isosceles Triangle"
          P 11     else
          P 12        Print "Scalene Triangle"
          P 13     endIf
```

Check Lengths are greater than zero

```
----------P-  2B  -P-----------------------------------------------
          P  1     if FIRST_LENGTH is greater than zero\
          P  2        and SECOND_LENGTH is greater than zero\
          P  3        and THIRD_LENGTH is greater than zero
          P  4        ...All Lengths are greater than zero
          P  5     else
          P  6        Print "Lengths must be greater than zero"
          P  7     endIf
```

Check that Lengths form a Triangle

```
----------P- 3A -P----------------------------------------

P  1    ...No triangle exists if one side is longer than the other
P  2    ...two sides together
P  3    if FIRST_LENGTH is longer then SECOND_LENGTH and THIRD_LENGTH
P  4      Print "Does not form a triangle"
P  5    endif
P  6    if SECOND_LENGTH is longer than FIRST_LENGTH and THIRD_LENGTH
P  7      Print "Does not form a triangle"
P  8    endif
P  9    if THIRD_LENGTH is longer than FIRST_LENGTH and SECOND_LENGTH
P 10      Print "Does not form a triangle"
P 11    endif
```

PROCEDURES

1A	1	Truss Type Analyser
1B	2	. Enter Lengths
1C	3	. Check Validity
2B	4	. . Check Lengths are greater than zero
3A	5	. . Check that Lengths form a Triangle
2A	6	. Display Type

PROCEDURE INDEX

PROCEDURE DEFINED
REFERENCED IN PROCEDURE
LINES

2B Check Lengths are greater than zero
1C Check Validity
1

3A Check that Lengths form a Triangle
1C Check Validity
2

1C Check Validity
1A Truss Type Analyser
2

2A Display Type
1A Truss Type Analyser
3

1B Enter Lengths
1A Truss Type Analyser
1

1A Truss Type Analyser

DATA ITEM
REFERENCED IN SEGMENT
LINES

FIRST_LENGTH
1B Enter Lengths
2
2A Display Type
1 2 7
2B Check Lengths are greater than zero
1
3A Check that Lengths form a Triangle
3 6 9

SECOND_LENGTH
1B Enter Lengths
3
2A Display Type
1 9
2B Check Lengths are greater than zero
2
3A Check that Lengths form a Triangle
3 6 9

THIRD_LENGTH
1B Enter Lengths
4
2A Display Type
2 7 9
2B Check Lengths are greater than zero
3
3A Check that Lengths form a Triangle
3 6 9

*** END OF DESIGN DOCUMENT ***

STATISTICS

PROCEDURE COUNT	:	6
PROCEDURE LINE COUNT	:	40
DATA ITEM COUNT	:	3
DATA ITEM USAGE COUNT	:	25
ERROR COUNT	:	0

****

END OF DESIGN DOCUMENT

****

STATISTICS

PROCEDURE COUNT	:	6
PROCEDURE LINE COUNT	:	40
DATA ITEM COUNT	:	3
DATA ITEM USAGE COUNT	:	23
ERROR COUNT	:	0

****

DESIGN LISTING

CURRENT REVISION

Time 17 : 34

Date 30 7 1988

****

Truss Type Analyser

1A

1B	P	1 Enter Lengths
1C	P	2 Check Validity
2A	P	3 Display Type

Enter Lengths

1B

	P	1 Print "Type in Length of Truss"
	P	2 Set FIRST_LENGTH to Number Read
	P	3 Set SECOND_LENGTH to Number Read
	P	4 Set THIRD_LENGTH to Number Read

Check Validity

1C

2B	P	1 Check Lengths are greater than zero
3A	P	2 Check that Lengths form a Triangle

Display Type

```
--------------P-  2A  -P------------------------------------------

If FIRST_LENGTH is equal to SECOND_LENGTH
    If FIRST_LENGTH is equal to THIRD_LENGTH
        Print "Equilateral Triangle"
    else
        Print "Isoceles Triangle"
    endif
elseif FIRST_LENGTH is equal to THIRD_LENGTH
    Print "Isoceles Triangle"
elseif SECOND_LENGTH is equal to THIRD_LENGTH
    Print "Isoceles Triangle"
else
    Print "Scalene Triangle"
endif
```

Check Lengths are greater than zero

```
--------------P-  2B  -P------------------------------------------

If FIRST_LENGTH is greater than zero
    and SECOND_LENGTH is greater than zero
    and THIRD_LENGTH is greater than zero
    ...All Lengths are greater than zero
else
    Print "Lengths must be greater than zero"
endif
```

Check that Lengths form a Triangle

```
----------P-  3A  -P----------------------------------------------

P  1    ...No triangle exists if one side is longer than the other
P  2    ...two sides together
P  3    if FIRST_LENGTH is longer than SECOND_LENGTH and THIRD_LENGTH
P  4       Print "Does not form a triangle"
P  5    endif
P  6    if SECOND_LENGTH is longer than FIRST_LENGTH and THIRD_LENGTH
P  7       Print "Does not form a triangle"
P  8    endif
P  9    if THIRD_LENGTH is longer than FIRST_LENGTH and SECOND_LENGTH
P 10       Print "Does not form a triangle"
P 11    endif
```

PROCEDURES

1A	1	Truss Type Analyser
1B	2	. Enter Lengths
1C	3	. Check Validity
2B	4	. . Check Lengths are greater than zero
3A	5	. . Check that Lengths form a Triangle
2A	6	. Display Type

PROCEDURE INDEX

PROCEDURE DEFINED	REFERENCED IN PROCEDURE	LINES
2B Check Lengths are greater than zero	1C Check Validity	1
3A Check that Lengths form a Triangle	1C Check Validity	2
1C Check Validity	1A Truss Type Analyser	2
2A Display Type	1A Truss Type Analyser	3
1B Enter Lengths	1A Truss Type Analyser	1
1A Truss Type Analyser		

DATA ITEM
REFERENCED IN SEGMENT
LINES

FIRST_LENGTH
1B Enter Lengths
2
2A Display Type
1 2 7
2B Check Lengths are greater than zero
1
3A Check that Lengths form a Triangle
3 6 9

SECOND_LENGTH
1B Enter Lengths
3
2A Display Type
1 9
2B Check Lengths are greater than zero
2
3A Check that Lengths form a Triangle
3 6 9

THIRD_LENGTH
1B Enter Lengths
4
2A Display Type
2 7 9
2B Check Lengths are greater than zero
3
3A Check that Lengths form a Triangle
3 6 9

****

DESIGN LISTING

CURRENT REVISION

Time 17 : 35

Date 30 7 1988

****

Examples from chapter 5

----------P- 1A -P----------

Example 1

----------P- 1B -P----------

```
P 1    ...This is a comment line
P 2    ...So is this and the next line
P 3    ...
```

Example 2

----------P- 1C -P----------

```
P 1    If NUMBER_ is one
P 2       print "you have chosen an apple"
P 3    elseif NUMBER_ is two
P 4       print "you have chosen an orange"
P 5    else
P 6       print "you may only enter a one or a two"
P 7    endif
```

Example 3

```
----------P- 2A -P----------

P 1    If the animal has fur
P 2        if the animal has a pouch
P 3            print "It may be a kangaroo"
P 4        endif
P 5    endif
```

Example 4

```
----------P- 2B -P----------

P 1    While the child is hungry
P 2        give it a biscuit
P 3    endwhile
```

Example 5

```
----------P- 2C -P----------

P 1    find door
P 2    While the door is locked
P 3    endwhile
P 4    open door
```

Example 6

```
----------P-  3A  -P----------------------------------------------------

P  1    repeat
P  2        add petrol to the tank
P  3    until the tank is full
```

Example 7

```
----------P-  3B  -P----------------------------------------------------

P  1    open the valve
P  2    repeat
P  3    until tank is full
P  4    close the valve
```

Example 8

```
----------P-  3C  -P----------------------------------------------------

P  1    case fruit
P  2    of orange , grapefruit
P  3        peel
P  4        eat
P  5    of grape
P  6        pick berries
P  7        eat
P  8    of apple
P  9        remove core
P 10        eat rest
P 11    default
P 12        print "only orange, grapefruit, grape or apple valid"
P 13    endcase
P 14    ?
```

Example 9

```
-------------P-  4A -P-----------------------------------------------

          P  1    Output (CARRIAGE_RETURN, LINE_FEED)
          P  2    ...enables feed of nutrients
          P  3    Open valve
```

Example 10

```
-------------P-  4B -P-----------------------------------------------

          P  1    Y_ = X_ + 1
          P  2    NEXT_VALUE = sin (X_ + 0,3)
          P  3    Print "this is a line"
```

Global variables

```
-------------D-  4C -D-----------------------------------------------

          D  1    ...This was Example 11
          D  2    ...The following is Example 12
          D  3    VALID_RECORD_POINTER used to indicate next valid record
          D  4    PERSONNEL_STARTDATE
```

PROCEDURES

1A	1	Examples from chapter 5
1B	2	Example 1
1C	3	Example 2
2A	4	Example 3
2B	5	Example 4
2C	6	Example 5
3A	7	Example 6
3B	8	Example 7
3C	9	Example 8
4A	10	Example 9
4B	11	Example 10

PROCEDURE INDEX

PROCEDURE DEFINED		REFERENCED IN PROCEDURE LINES
1B	Example 1	
4B	Example 10	
1C	Example 2	
2A	Example 3	
2B	Example 4	
2C	Example 5	
3A	Example 6	
3B	Example 7	
3C	Example 8	
4A	Example 9	
1A	Examples from chapter 5	

DATA INDEX

DATA ITEM
REFERENCED IN SEGMENT
LINES

CARRIAGE_RETURN
4A Example 9
1

LINE_FEED
4A Example 9
1

NEXT_VALUE
4B Example 10
2

NUMBER_
1C Example 2
4 3

PERSONNEL_STARTDATE
4C Global variables
4

VALID_RECORD_POINTER
4C Global variables
3

X_
4B Example 10
1 2

Y_
4B Example 10
1

DATA ITEM
REFERENCED IN SEGMENT
LINES

CARRIAGE_RETURN
4A Example 9
1

LINE_FEED
4A Example 9
1

NEXT_VALUE
4B Example 10
2

NUMBER_
1C Example 2
1 3

PERSONNEL_STARTDATE
4C Global variables
4

VALID_RECORD_POINTER
4C Global variables
3

X_
4B Example 10
1 2

Y_
4B Example 10
1

****

END OF DESIGN DOCUMENT

****

STATISTICS

PROCEDURE COUNT	:	11
PROCEDURE LINE COUNT	:	49
DATA ITEM COUNT	:	8
DATA ITEM USAGE COUNT	:	10
ERROR COUNT	:	0