# An Interactive COBOL Interpreter
# &
# JSP Debugger

Honours Project Report

By

Alan Richardson

BSc Computing 5

April 1992

# Contents

**1 Acknowledgements** **9**

**2 Preface** **10**

**3 Introduction** **10**

**I Background** **11**

**4 COBOL** **11**

**5 JSP** **13**
5.1 Control Constructs . . . . . 13
5.1.1 Sequence . . . . . 13
5.1.2 Selection . . . . . 14
5.1.3 Iteration . . . . . 15
5.1.4 The Design Process . . . . . 15
5.2 Advanced JSP Techniques . . . . . 16
5.2.1 Correspondence . . . . . 16
5.2.2 Inversion . . . . . 17
5.2.3 Backtracking . . . . . 18
5.2.4 Advantages of JSP . . . . . 19
5.2.5 PROBENCH . . . . . 19

**6 Compilers** **21**
6.1 Lexical Analysis . . . . . 22
6.2 Syntax Analysis . . . . . 24
6.2.1 Syntax Analysis & COBOL . . . . . 25
6.3 Code Generation . . . . . 25

**7 Interpreters** **27**

**8 Comparison of Compilers & Interpreters** **28**
8.1 Advantages of Compilers . . . . . 28
8.2 Advantages of Interpreters . . . . . 29

**9 Debugging** **30**
9.1 Debugging Tools . . . . . 30
9.2 Debugging Tools, COBOL and JSP . . . . . 31
9.2.1 A JSP Debugger . . . . . 32
9.3 Implications for JSP Development . . . . . 33

**II Approach** **35**

**10 Compiler Design** **35**
10.1 JSP, a compiler design methodology . . . . . 35
10.2 Mixing Data Types and Data Structures . . . . . 37

**11 JSP and 'C'** **37**
11.1 Inversion in 'C' . . . . . 38
11.1.1 Inversion in the Interpreter . . . . . 39

**12 Diagrammer Design** **40**

**13 Development of the Interpreter** **40**
13.1 Preparing the Symbol Table . . . . . 41
13.1.1 The Line Scanner . . . . . 41
13.1.2 The Scanner . . . . . 42
13.1.3 The Symbol Table Generator . . . . . 43
13.2 The Operations Table . . . . . 44
13.2.1 Scanning the Operations . . . . . 44
13.2.2 Parsing the Operations . . . . . 45
13.3 The Conditions Table . . . . . 45
13.3.1 Scanning the Conditions . . . . . 46
13.3.2 Parsing the Conditions . . . . . 46
13.4 The PROBENCH Diagram File . . . . . 47
13.5 Development of the Diagrammer . . . . . 47

**14 Testing** **49**
14.1 The Interpreter . . . . . 49
14.1.1 Testing the Symbol Table . . . . . 49
14.1.2 Testing the Operations . . . . . 50
14.1.3 Testing the Conditions . . . . . 50
14.2 The Diagrammer . . . . . 50
14.3 Testing the Debugger . . . . . 51

14.3.1 Multiplication . . . . . 51
14.4 Textual Loops . . . . . 52

**III Evaluation** **53**

**15 Evaluation** **53**
15.1 The Deliverables . . . . . 53
15.1.1 The Interpreter & The Diagrammer . . . . . 53
15.1.2 The Code Translator . . . . . 54
15.2 The Approach . . . . . 54
15.2.1 JSP a Compiler Design Methodology . . . . . 54
15.2.2 Mixing Data Types & Data Structures . . . . . 56

**16 Problems & Setbacks** **56**
16.1 Problem 1: Unproductive Development Time . . . . . 57
16.2 Problem 2 : Lack of Inversion . . . . . 57
16.3 Problem 3 : Memory Intensive Data Structures . . . . . 58
16.4 Problem 4 : File Incompatibility . . . . . 58

**17 Future Work** **58**
17.1 Enhancements . . . . . 58
17.2 Future Developments . . . . . 60
17.2.1 Keyword Recognition . . . . . 60
17.2.2 Hooks into Probench . . . . . 60
17.2.3 Symbol Table Handling . . . . . 61
17.2.4 A Fully Functional JSP Debugger . . . . . 61
17.2.5 Removal of Redundant Code . . . . . 61
17.2.6 Enhancements to the Diagrammer . . . . . 62

**18 In Conclusion** **62**

**IV Appendices** **67**

**A COBOL Interpreter User Guide** **67**
A.1 '.dat' File Conventions . . . . . 67
A.2 '.ops' File Conventions . . . . . 67
A.2.1 MOVE . . . . . 67
A.2.2 ADD . . . . . 67

A.2.3 DISPLAY . . . . . 68
A.3 '.cnd' File Conventions . . . . . 68
A.4 Preparing for Interpretation . . . . . 68

**B Diagrammer User Guide** **70**
B.1 Commands . . . . . 70
B.2 Debugging with the Diagrammer . . . . . 71
B.3 Screenshots . . . . . 72

**C Data Structures** **75**
C.1 The Symbol Table . . . . . 75
C.2 The Operations Table . . . . . 76
C.3 The Conditions Table . . . . . 77
C.4 The Stack . . . . . 78
C.5 The Trie . . . . . 79
C.6 The Diagrammer Data Structures . . . . . 81
C.6.1 The Tree . . . . . 81
C.6.2 The Boxlist . . . . . 82
C.6.3 The Oplist . . . . . 83
C.6.4 The Box . . . . . 84

**D Test Programs** **85**
D.1 Multiply . . . . . 85
D.1.1 Operations . . . . . 85
D.1.2 Conditions . . . . . 85
D.1.3 Data Definitions . . . . . 85
D.1.4 Diagram . . . . . 87
D.2 Text Loop . . . . . 88
D.2.1 Operations . . . . . 88
D.2.2 Conditions . . . . . 88
D.2.3 Data Definitions . . . . . 88
D.2.4 Diagram . . . . . 89

**E COBOL to C** **90**
E.1 Necessity . . . . . 90
E.2 Design . . . . . 91
E.3 Development . . . . . 92
E.4 User Guide . . . . . 95
E.4.1 The Data Definition . . . . . 95

E.4.2 Conditions . . . . . 96
E.4.3 Operations . . . . . 96
E.4.4 Quits . . . . . 96
E.4.5 Using The Program . . . . . 97
E.5 Conversion Strategies . . . . . 98
E.5.1 Comments . . . . . 98
E.5.2 Procedure Division . . . . . 98
E.5.3 STOP RUN . . . . . 98
E.5.4 GO TO lines . . . . . 98
E.5.5 Labels . . . . . 98
E.5.6 ELSE . . . . . 98
E.5.7 IF & IF NOT . . . . . 98
E.5.8 Conditions . . . . . 99
E.5.9 Quits . . . . . 99
E.5.10 Normal Lines . . . . . 99
E.6 JSP Design . . . . . 100
E.6.1 Operations . . . . . 100
E.6.2 Conditions . . . . . 101
E.6.3 Quits . . . . . 102
E.6.4 Data Definitions . . . . . 102
E.6.5 The Diagram . . . . . 106
E.6.6 Code . . . . . 130

**F The Interpreter JSP Designs 181**
F.1 Scanning The '.dat' . . . . . 181
F.1.1 Operations . . . . . 181
F.1.2 Conditions . . . . . 181
F.1.3 Data Definitions . . . . . 181
F.1.4 The Header File . . . . . 182
F.1.5 The Diagram . . . . . 184
F.1.6 The Code . . . . . 191
F.2 Tokenising The '.dat' . . . . . 209
F.2.1 Operations . . . . . 209
F.2.2 Conditions . . . . . 210
F.2.3 Data Definitions . . . . . 210
F.2.4 The Diagram . . . . . 212
F.2.5 The Code . . . . . 217
F.3 Building The Symbol Table . . . . . 233
F.3.1 Operations . . . . . 233

F.3.2 Conditions . . . . . 234
F.3.3 Data Definitions . . . . . 234
F.3.4 The Diagram . . . . . 236
F.3.5 The Code . . . . . 242
F.4 Tokenising The '.ops' . . . . . 260
F.4.1 Operations . . . . . 260
F.4.2 Conditions . . . . . 261
F.4.3 Data Definitions . . . . . 261
F.4.4 The Quits . . . . . 262
F.4.5 The Diagram . . . . . 263
F.4.6 The Code . . . . . 269
F.5 Parsing The '.ops' . . . . . 287
F.5.1 Operations . . . . . 287
F.5.2 Conditions . . . . . 287
F.5.3 Data Definitions . . . . . 288
F.5.4 The Diagram . . . . . 289
F.5.5 The Code . . . . . 291
F.6 Scanning The '.cnd' . . . . . 299
F.6.1 Operations . . . . . 299
F.6.2 Conditions . . . . . 300
F.6.3 Data Definitions . . . . . 300
F.6.4 The Quits . . . . . 301
F.6.5 The Diagram . . . . . 302
F.6.6 The Code . . . . . 309
F.7 Parsing The '.cnd' . . . . . 329
F.7.1 Operations . . . . . 329
F.7.2 Conditions . . . . . 330
F.7.3 Data Definitions . . . . . 330
F.7.4 The Diagram . . . . . 332
F.7.5 The Code . . . . . 337

**G C Code** **350**
G.1 Interpreter Source Code . . . . . 351
G.1.1 Symbol Table . . . . . 351
G.1.2 Operations Table . . . . . 377
G.1.3 Conditions Table . . . . . 391
G.1.4 Stack By Array . . . . . 402
G.1.5 Trie . . . . . 405
G.1.6 Intermediate File Tokens . . . . . 409

G.1.7 Keyword Definitions . . . . . 414
G.1.8 Trace Functions . . . . . 418
G.2 Diagrammer Source Code . . . . . 426
G.2.1 Diagrammer Structures . . . . . 426
G.2.2 Boxlist . . . . . 428
G.2.3 Boxes . . . . . 436
G.2.4 Operations . . . . . 443
G.2.5 Tree Structure . . . . . 445
G.2.6 Tree Manipulation . . . . . 447
G.2.7 Diagrammer Functions . . . . . 459
G.2.8 Drawing Functions . . . . . 472
G.2.9 The Diagrammer Program . . . . . 475
G.3 Probench '.pr0' File Handling . . . . . 476
G.4 General Code . . . . . 486
G.4.1 File Handling . . . . . 486
G.4.2 Error Handling . . . . . 489
G.4.3 String Conversion . . . . . 491
G.5 Redundant Code . . . . . 492
G.6 Trie Input Data . . . . . 496

**H Project Management Information** **505**
H.1 Gantt Chart . . . . . 505
H.2 Project Diary . . . . . 506

# List of Figures


1 JSP Sequence Constructs . . . . . 14
2 JSP Selection Components . . . . . 14
3 JSP Iteration Constructs . . . . . 15
4 JSP Backtracking Constructs . . . . . 18
5 the process of compilation . . . . . 21
6 Compilation steps . . . . . 21
7 Complete Lexical Analysis . . . . . 23
8 Incomplete Lexical Analysis . . . . . 23
9 The Symbol Table . . . . . 75
10 The Operations Table . . . . . 76
11 The Conditions Table . . . . . 77
12 The Stack By Array . . . . . 78
13 The Trie . . . . . 79
14 The Tree . . . . . 81
15 The Boxlist . . . . . 82
16 The Oplist . . . . . 83
17 The Box . . . . . 84
18 Project Gantt Chart . . . . . 505

# 1 Acknowledgements

I would like to acknowledge the help and assistance of the following people who have contributed to the project.

**John Pate** for the Probench file information.

**Ralph Storer** for suggesting that I use Probench files.

**Craig Goldie** for his LaTeX & xpic information.

**Glyn Gardiner** for being the original sounding board.

And everybody else who thought it was a good idea.

# 2 Preface

The format of the report has been split into four parts.

The background information which gives a general overview of compilers, interpreters, COBOL and Jackson Structured Programming. This part also discusses debugging and the reasons behind the development of the project.

Part 2 discusses the techniques chosen to develop the project.

The third part summarises the report contents, describing the finished product and evaluating the techniques used. Further development ideas are discussed and overall conclusions are drawn.

Part four contains the appendices, the various user guides, descriptions of the data structures used and JSP designs. In an effort to provide as complete documentation to support the project, the code generated from the JSP designs have been included. This has unfortunately increased the size of the document but had this been omited then the resulting documentation would have been incomplete.

# 3 Introduction

This project resulted from a dissatisfaction with the current JSP design environments available. Mainly their lack of support for on-line testing.

This project required the construction of an interactive COBOL interpreter coupled with a JSP debugger. This was intended to be interfaced with Probench, an existing JSP diagrammer, unfortunately the interface was not developed. As a result the interpreter has been interfaced with a diagrammer developed during the course of the project.

The debugger allows the diagram to be run interactively, with the progress of the run being displayed. The variables and their contents are also available for interrogation at any point by the user.

The project also attempts to evaluate the use of JSP as a compiler design methodology and the benefits which could be gained from the use of a JSP debugger.

# Part I
# Background

## 4 COBOL

The COmmon Business Oriented Language (COBOL) is often malinged but consistently chosen as *the* language for the development of data intensive applications.

> *It has been indicated that there are over 75 billion lines of COBOL code in existence. Recent surveys also indicate that over 90% of the world's commercial development is performed using COBOL.* [McCRORY 92]

COBOL's continued longevity in the face of 'better' programming languages owes much to the mass of COBOL code already written and being maintained. But also to the number of proven design techniques and methodologies applicable to it, the most notable being Jackson Structured Programming (JSP).

Over the 30 years of its existence COBOL has undergone numerous changes which have helped to make it one of the most flexible languages available for commercial product development. The language standardisations carried out by the American National Standards Institute (ANSI) resulting in COBOL 74, COBOL 85, and possibly COBOL 9X [1].

Its popularity has contributed to a vast array of third party products available to aid COBOL development:- report generators, screen designers, code generators, and reverse engineering tools being the most commonly found examples. Less common products are COBOL pre-compilers which allow *#include* type directives and *#define*'s. This is comparable with the 'C' languages directives to define constants, and include other header files.

Databases are often used in conjunction with COBOL programs in the business world, and Structured Query Languages (SQL) can be embedded within many COBOL language implementations. This allows the procedural complexity of COBOL to be used with the data retrieval and storage mechanisms of databases. The Ryan-McFarland COBOL-85 implementation from Liant even allows transparent access to the RM/plus Database system.

[1] currently being examined by the ANSI X-3J4 committee

The design of the COBOL language has concentrated on providing language features applicable to Data intensive file handling applications. Designed to be readable enough for programs to be self documenting; the large number of English language keywords and the pseudo natural language syntax , tend to make code voluminous. Obvious benefits of this are that beginners can learn to program fairly quickly; and existing code can be understood easily, although badly designed code in COBOL is large and unwieldy.

A COBOL program is split into separate divisions, thereby removing the machine dependant functions from the machine independent functions. This allows the development of portable programs which need only be changed in the ENVIRONMENT division (the machine specific descriptions). The DATA division and PROCEDURE division, remain unchanged. These two divisions also serve to separate the data descriptors from the procedural aspects.

# 5 JSP

In the '70's when Jackson first presented his ideas of a structured programming methodology the two main methods of program design were flow charts and structured programming.

Flow charts are a documentation tool to describe the flow of control around a program graphically. To quote Jackson

> *At the time when the designer should be thinking about the shape of his problem, the flowchart encourages him to think about execution of the program inside the computer. Of course, some programmers who draw flowcharts do produce well structured programs; but their success is in spite of the technique, not because of it.* [JACKSON 75]

Structured Programming, normally attributed to Dijkstra, advocates 3 constructs, sequence, condition and repetition [PRESSMAN 87]. Also an avoidance of GOTO's and the use of block structures.

The problem here being that Structured Programming was a set of guidelines intended to make programs more robust and maintainable, but they did not help the programmer to design their programs. This led to a proliferation of invented constructs which were not apparently derived from the problem but simply to use the structured prgramming philosophy of modular code and goto avoidance. This in turn led to bad but perfectly *structured* programs being written.

Jackson drew upon the ideas of structured programming to produce his methodology. The basis of the methodology being Stepwise refinement, and restricting the set of control constructs used.

Stepwise refinement is the process of taking the stated program function and splitting it into smaller more detailed functions which are themselves broken down, until the program operation level is reached.

## 5.1 Control Constructs

The three constructs are sequence, selection and iteration.

### 5.1.1 Sequence

A component consisting of two or more sub-components which occur once each in order. An example is shown on figure 1 .

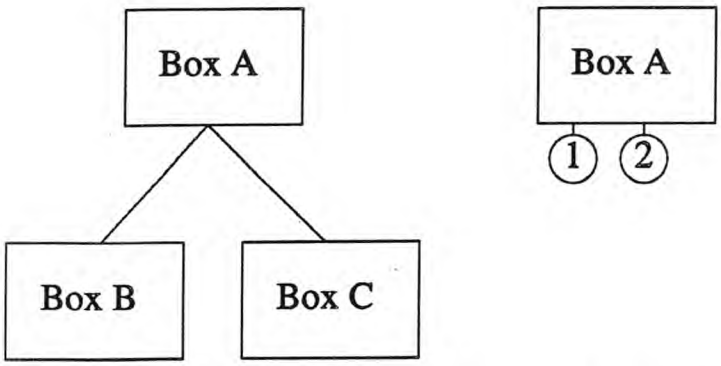



Figure 1: JSP Sequence Constructs

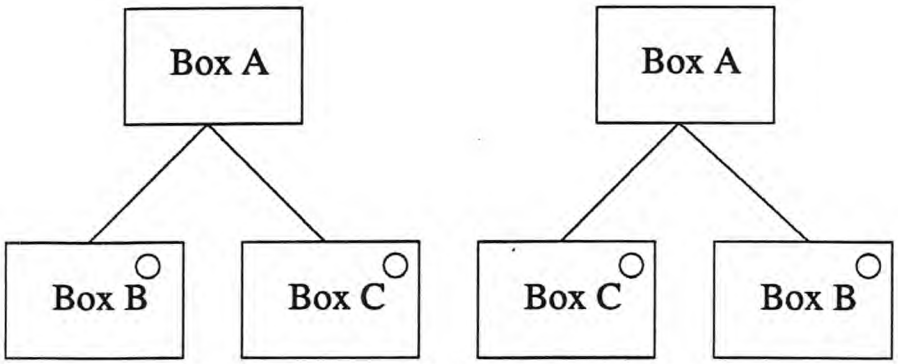



Figure 2: JSP Selection Components

From the diagrams the process of stepwise refinement can clearly be shown to be in operation. Function Component 'A' can either be a sequence of function components or can be a sequence of operations.

### 5.1.2 Selection

A component that has two or more parts of which one and only one occurs once, an example is shown on figure 2.

The actual ordering of the components is structurally unimportant as both diagrams have structural equivalence. When programming this construct, care must be taken to ensure that the conditions allocated to the sub-components are specific enough to guarantee this equivalence.

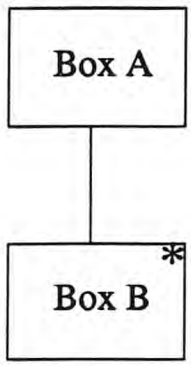



Figure 3: JSP Iteration Constructs

### 5.1.3 Iteration

An iterated component has only one sub-component which can occur zero or more times. This is illustrated on figure 3.

### 5.1.4 The Design Process

The basic design process of JSP follows 3 steps:

> 1. *consider the problem environment and record our understanding of it by defining structures for the data to be processed;*
> 2. *form a program structure based on the data structures;*
> 3. *define the task to be performed in terms of the elementary operations available, and allocate each of those operations to suitable components of the program structure.*
>
> [JACKSON 75]

**Step 1, Data Analysis -** The first stage of JSP design is to analyse the data which is to be processed. This includes all the input data formats and output data formats. These are all translated into data structure diagrams using the constructs shown in figures 1, 2 & 3.

**Step 2, Produce Program Structure -** The program structure initially looks like the data structure diagrams. The names are changed to

reflect the processing aspects. For the case of programs with multiple data structures a technique called correspondence is used to combine them, see section 5.2 on page 16.

**Step 3, Allocate Operations and Conditions -** The program specification is examined and the detailed processing necessary to produce the output data from the input data is considered. This analysis is used to produce a list of operations and conditions, these are written informally at first and when the development proceeds they are written in the implementation language.

Once all the steps have been followed the program has been designed and the final code can be produced. This can be done in 3 ways.

1. transfer the design to a diagrammer and generate the code.
2. convert the design into programming language statements.
3. convert the design into a program description language such as Jackson's schematic logic [JACKSON 75] or HOST [STORER 87].

If the diagram is being manually coded then Jackson recommends using the 3rd step.

## 5.2 Advanced JSP Techniques

The previously outlined steps make up the basic design technique as presented by Jackson. These are adequate for simple programs processing one data structure.

More complicated programs require more advanced techniques; correspondence and inversion for combining multiple structures and backtracking for handling difficult selections.

### 5.2.1 Correspondence

Correspondence combines two structures into one, matching together the corresponding components, and any non-corresponding components are merged into the diagram without change.

Correspondence relies on the ability of the designer to spot similarities between the structures to be processed. It is an informal technique and is

often difficult to learn, but for simple programs it can be quick and produces simple designs.

There are cases when this technique can not be used. These cases arise when the data structures conflict or clash. Three types of structure clash have been identified.

1. Ordering clash
2. Boundary clash
3. Interleaving clash

An **Ordering Clash** occurs when the data structures to be combined use essentially the same structures but occur in different orders.

A **Boundary Clash** occurs when the low level components occur under different circumstances. To illustrate this, consider the following example [STORER 87]. An input file containing departments and a number of employee records belonging to those departments is to be printed on an output report. The output format is 10 employee records per page with a department header at the top of each page. These two structures do not correspond because although there are 10 lines to a page there may be more or less than ten employee records to a department. The boundaries of page and department do not coincide.

An **Interleaving Clash** is a form of ordering clash where a specific item has a number of components which exist in sequence on a file but may be interspersed with other items. For example, a source file contains the following data:

1.START, 2.START, 3.START, 2.BODY, 1.BODY, 2.END, 3.BODY, 3.END, 1.END

The lower level structure sequence of an item (START, BODY, END) does not correspond with the file containing the *iteration* of items.

There are techniques for getting around these clashes but the simplest way is to not encounter them at all, and inversion helps the programmer do just that.

### 5.2.2 Inversion

Inversion keeps each data structure separate and forms a program structure from each data structure. Operations and conditions are allocated to allow the structures to execute and then they are linked together.

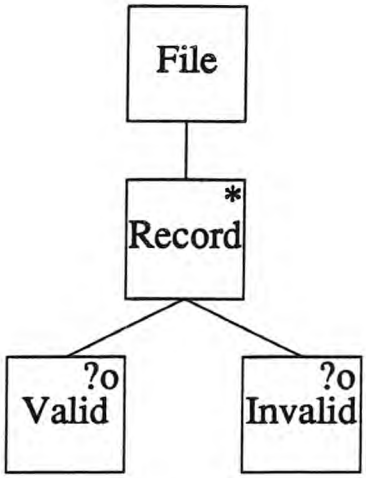



Figure 4: JSP Backtracking Constructs

At first this process involves using intermediate files to communicate between the processes. This has the advantage that each program can be tested in isolation thereby resulting in a more robust system when they are inverted.

In order to invert the programs, the communication between them must be considered. The programs are viewed as co-routines each having its own internal state and they GET data from one process, formerly an intermediate file, and PUT data to another, again formerly an intermediate file.

This can be carried out between the programs themselves or a program known as a scheduler can be created to invoke the co-routines when necessary.

### 5.2.3 Backtracking

Program design techniques often use "read ahead" in order to evaluate conditions. This is commonly used to evaluate EOF. However, there are some circumstances where the reading ahead of a record or character is insufficient information to evaluate a condition. This can be solved by using either "multiple read ahead" or backtracking.

A problem which cannot be solved by multiple read ahead is when a condition to be evaluated, cannot be evaluated because it depends upon the results of data processing which has not yet been carried out. This problem can be solved by backtracking. A common example is that of a file of records some of which a valid and others are invalid as figure 4 shows.

The record is assumed to be valid, but if at any point in the processing

of the valid record it is discovered to be invalid then a quit is issued and control is passed to the invalid component.

The valid component is known as the POSIT while the invalid is known as the ADMIT.

### 5.2.4 Advantages of JSP

- It is a formalised design method, therefore it can be taught. It it less reliant on experience than other methods, making it suitable for both the beginner and the experienced designer. This suggests that designs produced using JSP by different programmers are similar, making it easier for other programmers to understand someone else's work.
- An understanding of JSP provides a greater insight into programming than structured programming methods alone.
- Reliability is improved since the approach demands that ambiguities and omissions be revealed at early stages of the design process.
- Documentation is an integral part of the method, thereby increasing communication and leading to more accurate and complete system documentation. This does not imply that the method is self documenting, it simply means that more useful documentation is available than code alone.
- It can be automated, this allows more time to be spent on the design because the design is also the code. It also means that maintenance is easier because all changes are made at the design level, leaving it less open to "quick fixes".

### 5.2.5 PROBENCH

Probench is a software tool which allows the creation and editing of JSP diagrams. This makes the process of JSP design more flexible and efficient.Probench generates COBOL code and this reduced effort in coding offers a substantial decrease in program development time.

The diagrammer also provides integrity checking of diagrams which can help to prevent simple mistakes being carried across into the code.

Probench provides a debugging tool called trace which can provide a listing of the paths taken by a program when it is run. The current project

aims to improve on this debugging aid, the need for improvement is explained in section 9 on page 30

# 6 Compilers

Although the current project entailed the development of an interpreter (discussed in the next section) it is necessary to first discuss compilers, upon whose techniques an interpreter is based.

A compiler is a program which takes as input the high level source description of another program, and produces as output object code that can be executed by the machine.

Figure 5: the process of compilation

The process by which a compiler achieves this is essentially to carry out the following steps.

1. Lexical Analysis (Scanning)
2. Syntax Analysis (Parsing)
3. Code Generation

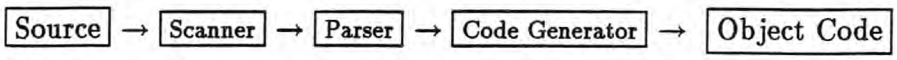



Figure 6: Compilation steps

These steps are carried out in order and any source errors are reported at the relevant stage. Simple syntax errors are found during lexical analysis. Syntax Analysis finds the more detailed syntax errors, and during Code generation, linkage errors and undefined functions are identified.

## 6.1 Lexical Analysis

The input source supplied for compilation consists of a stream of alphanumeric characters that are understandable by the human programmer. The lexical analyser breaks down the source into symbols or tokens that can be understood by the syntax analyser.

To do this the lexical analyser must determine which characters are significant, removing comments, blank spaces and spurious line breaks. The significant code is then scanned to determine where the tokenisable portions begin and end. These two processes are often done simultaneously.

There are a variety of symbols used to represent scanned data :non-numeric literals, operators, numeric literals, reserved words, and delimiters.

**Non-numeric literals** or string literals such as "Compilers compile, Interpreters interpret" tend to be easily recognisable because they are delimited by quotes ',',", and " in COBOL a string delimited by ', and ' may contain ", and " as characters and vice versa.

**Operators** such as ">","=<" and "! =" are normally referred to as specials since special handling has to be given in order to distinguish an "=" from a "=<".

**Numeric literals** are found because they consist purely of numbers although the end of the number may be harder to determine. This is true in COBOL where "123." could be an incorrect real number such as "123.5" or a number at the end of a statement. The actual context of where it is used determines its nature but the lexical analyser gives no thought to context and takes it as it comes. So both of the above examples are likely to be parsed in the same way : "123",".".

**Reserved words** and **identifiers** are distinguished because the list of keywords is already known. This becomes more difficult in a language like Fortran where it is permissible to have identifiers overloading the reserved word names, allowing variables called IF and ELSE. This can lead to parsing difficulties with some statements:-

```
IF IF=ELSE
THEN IF=ELSE-1
ELSE ELSE=IF
ENDIF
```

**Delimiters** such as ".","," , and ";" tend to be recognised immediately and their meaning is found by analysing the context in which they occur.

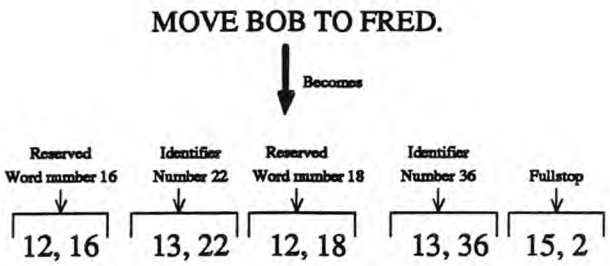



Figure 7: Complete Lexical Analysis

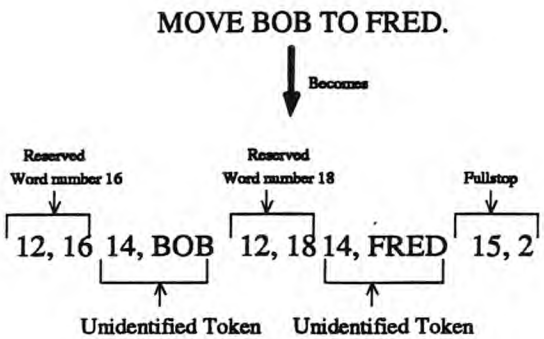



Figure 8: Incomplete Lexical Analysis

The lexical analyser does as much as it can without knowledge of the detailed syntax or semantics of the language and produces a table of non-numeric literals, a compressed list of instructions and possibly a symbol table of identifiers, shown in figure 7

The identification of BOB and FRED as identifier names may not be done at this stage, resulting in the list shown in figure 8

14,BOB and 14,FRED being used to indicate that BOB and FRED are unrecognised character sequences. Their context will be analysed at a later stage to find out what they represent.

This is a simplified example since an identifier, may share the same name as another identifier and some means must be available to distinguish them

from each other.

## 6.2 Syntax Analysis

The process of syntax analysis involves taking the stream of symbols from the lexical analyser and analysing them in context.

The context is found by using the predefined grammar of the language. In COBOL this is informally described using a subset of Backus Naur Form(BNF) see section 6.2.1.This can be done in two ways, bottom-up or top-down.

> **Bottom-up** analysis looks for patterns in the input which correspond to phrases in the grammar. The grammar shows how items can be combined into phrases, how phrases and items can be combined into larger phrases. The analyser therefore replaces first simple patterns of items by a single phrase-description (a pointer to a tree node), then more complex patterns of items and phrase-descriptions by a phrase-description, until the entire input has been compressed into a description of a <statement> phrase.
>
> **Top-down** analysis assumes that the input will be a statement. The grammar shows how a <statement> phrase can be built up of sub-phrases, and to find a statement the analyser merely searches for each of the sub-phrases in turn. To find a sub-phrase the analyser searches for each of its (sub-)sub-phrases, and so on until the search is reduced to a matter of searching for a single item, which it looks for in the input.
>
> [BORNAT 79]

The syntax analysis procedure is very often the driving force behind the lexical analyser. The lexical analyser scans some source code until a token has been identified. The token gets passed back to the syntax analyser which processes it and then calls the lexical analyser for another token.

Syntax analysis builds and maintains all the tables necessary for compilation, such as the symbol table, statement lists,and condition tables. Assuming that no syntax errors have been found then code generation is carried out.

### 6.2.1 Syntax Analysis & COBOL

> *Block-structured languages whose syntax can be defined in straightforward terms using a formal language definition such as BNF can be handled using fairly standard approaches. Languages such as FORTRAN and COBOL which have little formal syntactic structure tend to require quite different approaches to syntax analysis for each statement type, and much of the syntax analysis may be done by informal methods.*[MADDIX 89]

As [MADDIX 89] mentions no formal definition such as Backus Naur Form exists to describe the COBOL language. COBOL uses a variant of BNF which simplifies the description of the language syntax, but does so less rigorously. The following example is taken from [PRATT 84] and shows the syntactic statement "a signed integer is a sequence of digits preceded by an optional plus or minus".

- BNF

```
<signed integer>::= + <integer>| - <integer> | <integer>
      <integer>::= <digit> | <integer> <digit>
```

- COBOL description

```
<signed integer> ::= [+ -]  <digit> ...
```

The COBOL definition is simpler but BNF allows for a recursive definition to be coded from the specification. The COBOL definition leaves the implementation and interpretation up to the programmer.

## 6.3 Code Generation

When a program has been identified as syntactically correct, the set of tokens are mapped onto either machine instruction codes which can be executed at a later date, or onto assembly language statements which must be passed through an assembler before being run.

The object code produced in either of these methods, may not be capable of being run directly. It may need to be linked with other modules to form the complete program. This offers a very flexible method of program development since each module can be compiled separately. Then linked together once each has been tested in isolation.

The disadvantage with this being that the process of linking the modules together can be quite complex. Utilities such as MAKE on the UNIX system have been designed to help alleviate these problems. Some compilers, for example TURBO C, use timestamping to determine which, if any, of the constituent source and object codes have been changed and need either re-compilation or re-linking.

The compiler may also optimise the code produced. This means that there is no longer a simple mapping between the source and the object code. The compiler uses a number of different strategies to optimise for memory usage, speed and code size.

# 7 Interpreters

An interpreter executes a program written in a given source language by examining each statement in turn and performing the required action. For example, consider the following statement

```
ADD 2 TO SUM-COUNT.
```

The interpreter scans this statement recognising it as an addition, finds SUM-COUNT in the internal symbol table and uses the code contained within the interpreter to increase the value of this variable by 2.

Interpreting a statement is a long and complex process when compared with the equivalent compiled code. The statement has to be scanned and analysed every time it is encountered, symbols have to be retrieved and then the action of the statement must be carried out. This may take hundreds of machine instructions whereas the compiled version could be done as follows.

```
add #2,WORD PTR SUM-COUNT.
```

This means that an interpreter runs slower than a complier, often 10 or even 100 times slower.

Essentially the interpreter carries out the same jobs as a compiler in that it scans the code and checks for syntax errors. The syntax checking can be done in two ways by an interpreter.

- before interpreting the code, for example whilst typing in the source.
- or whilst interpreting.

The second option allows code to be altered interactively, but also means that syntax analysis will be done each time the statement is encountered. An added danger with this approach is that syntax errors may lie dormant in some unexplored regions of the code which are not tested until well into development.

Interpreters often use a very simple approach to syntax analysis and statement evaluation. For example, most BASIC interpreters consist of an initial scanning routine which identifies the type of statement normally by looking at the first keyword i.e. PRINT, or INPUT. The appropriate routine to be called may then be done using a CASE statement. Interpreters tend to have specific routines to handle each keyword. More general routines will be available to handle conditions.

Some language interpreters allow variables to be used without being declared e.g a BASIC interpreter when it encounters

```
10 X = 10
```

will place X in the symbol table and allocate it the value 10. This can only be done if the syntax of the language permits it. COBOL, for example, does not and requires all identifiers to be declared in the WORKING-STORAGE section.

# 8 Comparison of Compilers & Interpreters

## 8.1 Advantages of Compilers

- Compilers provide greater run-time efficiency than interpreters, with execution times often being 100 times less. Of course this does depend upon the type of application. For an I/O intensive program both can seem to be performing at the same speed but this is purely because the speed of the I/O devices acts as a drag factor.

- Compiled code uses less memory than interpreted code. Interpreted code needs the interpreter and source code in memory at the same time. Compiled code needs only the object code version of the program. This also means that an interpreted program can not be run without the interpreter and it is interpreted at source level so every developer using an interpreted language gives their code away with the product.

- Compilers provide a flexibility of development not normally associated with interpreters. This is achieved through the use of modular code which can be compiled separately and linked together. Interpreters which do provide this type of feature normally require all the modular source code to be held in memory at the same time.

- Compilers syntax check the entire source code before producing the final program. Most interpreters syntax check the code line by line, this means that some errors can lie dormant in some rarely executed piece of code.

## 8.2 Advantages of Interpreters

- Interpreters are particularly useful for debugging. Any exceptions, such as a divide by zero, would be trapped by the interpreter. The interpreter could then report the error, and the statement which generated the error. This has become less true now that compilers can provide source level debugging, e.g. MF-COBOL . But debugging can usually be carried out at a faster rate within an interpreter. See also section 9

- Using a compiler involves the lengthy process of compile, link, and then loading the object code. Some compilers may even force the programmer to use an assembler to produce the final executable. With an interpreter the process is simple, just load and go.

- Interpreters are potentially machine independent if they are written in a portable high level language. Compilers have to be rewritten to produce the object code needed by the target machine.

- 4GL's are more likely to be interfaced with an interpreter to allow incremental development, with a compiler only being used on the final version of the product.

# 9 Debugging

Debugging is an important part of any project development, whether it be done using JSP, Structured Programming, Compilers or Interpreters. To take the development of a product using JSP as an example, the typical development proceeds as follows; design, generate code, error and then debug.

The idea of debugging is the same no matter which methodology is used; to remove all bugs from the program. The actual manner in which this is done should be slightly different for each methodology. In the case of JSP, debugging should be done on the design; with all changes being made to the diagram, and the code *then* being either re-generated or modified.

Because the essential processes of debugging are the same no matter which method of development is used, an experienced programmer tends to be better at removing errors than a novice.

JSP as a design method is suitable for both novices and experienced designers, but the debugging of these programs will be done better by the more experienced programmer, even if they are novices to the method.

[OMAN 89] states that *"the success rate of the novice deteriorates in the absence of clues"* It is important then to give a novice as much help as possible when debugging.

The debugging tools available tend to depend upon the language implementation chosen, rather than the design method used. It is generally accepted that interpreters provide better debugging environments than compilers, although this is becoming less true as compilers improve.

## 9.1 Debugging Tools

The main debugging tools are monitors & disassemblers, source level debuggers and an interpreted language with debugging aids.

**Monitors & disassemblers** are suitable only for the most experienced programmers since they show the object code which the computer uses in the form of assembly language. Difficulties can arise from using this to debug programs written in a high level language since the compiler does not generate as efficient code as a human. Compiled code uses the internal stack register more extensively than a human programmer and this can make code difficult to follow. It is really only useful when all other debugging methods fail and the fault is a compiler error.

**Source level debuggers** are being supplied more and more often with compilers. The compiler usually has to generate extra code in order to use

them but they can be very helpful when debugging programs. Traces can be set up on variables, breakpoints can be set on procedures and lines, any errors are trapped and control is returned to the user at approximately the location where the error occurred.

**Interpreted languages** are a natural debugging environment especially if extra facilities such as trace, and variable monitoring have been included. The main benefit is that the programmer can interrogate the debugger using the language which is being interpreted. So variables can be DISPLAYed at any time, and MOVE's can be carried out during single stepping to alter variable values. Under this environment changes can be made to the source immediately and their effect tested without the need for recompilation.

## 9.2 Debugging Tools, COBOL and JSP

The aforementioned debugging tools are those that are typically used during any project development. The question that needs to be answered is, **are these tools adequate for development in JSP and COBOL?**

The JSP development cycle mentioned before, is:- design, code, error, debug. Debugging would probably be done using a source level debugger on a compiler since there are very few interpreted COBOL environments available.

The process of debugging will first be considered from the point of view of the novice, because the novice derives the most benefit from using a debugger.

Assuming that the novice can use the source level debugger adequately, they trace the program and discover the bug. The code is altered, re-run and then another bug is found ..., this continues until all the bugs have been removed.

The problem, is that the novice has been debugging at source code level, the temptation then is to fix the bugs, at source level. Once debugging is complete the novice then tries to alter the design to reflect changes made to the code . This can lead to some bugs remaining unchanged on the design, and the user can have a great deal of trouble getting their program to work.

It has been assumed that only the novice does this. The same temptations exist for the experienced programmer. They may believe that they know what the error is, follow up their suspicions using the code, and *then* alter the design. The experienced designer just tends to be better at keeping code and design in sync with each other.

Even if neither the novice or the experienced designer succumbs to the

temptation of source code editing, there are problems using the current technology in the correct manner. The continual movement from diagrammer, to compiler becomes time consuming and annoying. This can often discourage even the best intentioned JSP designer, and have them editing the code rather than return to the diagrammer.

### 9.2.1 A JSP Debugger

The most obvious fallacy that exists in the previous description of JSP debugging is that both the novice and the experienced designer have been using source level debuggers. They haven't, the source in a JSP design is the diagram. A source level debugger for JSP would allow the programmer to debug at the diagram level.

Debuggers like this would work in a similar way to typical source level debuggers. The main difference would be the level of detail available, not only would the operation currently being executed be shown on screen, the position of this operation on the diagram would be highlit as would the box that it has been allocated to.

Running the program would involve all operations accessed being shown in text form in some window, with boxes being lit and unlit as progress was made around the diagram. The user interface would be more intuitive than the associated source level debugger on a compiler because of the graphical interface. Boxes could be clicked on with the mouse to set and remove breakpoints, icons could be used to single step and run to completion, all other operations would be easily accessible from pull-down or pop-up menus. This is a far cry from learning numerous control keys, and a special debugging language to 'SHOW ALL VARIABLES'.

A diagram level debugger would allow the designer to track down errors using the design, thereby removing the temptation to edit the code. It also removes the need to look at the code from a different perspective.

This different perspective is used every time the programmer moves from the diagram to the code. The programmer has not been thinking about the problem in the same way that they would have been, had they been coding it directly. They have been thinking about the program in terms of actions on data, and when the code is being examined they are forced to think of the program in a procedural manner.

If the code has been generated then this problem *increases* because they did not write the code that they are debugging. The code produced has been written using their design, but effectively by someone else, using coding

conventions that they undoubtedly would not have used. Therefore before they can start to debug, they must first understand code written by another programmer. This places them in a maintenance role as well as a design role.

## 9.3 Implications for JSP Development

A JSP debugger could help the novice programmer in a number of ways :

- There is no need to see or understand the code produced. The emphasis is then placed on understanding the design techniques used to produce a program. Thereby making them more experienced and conscientious developers, concentrating more on design than coding.
- As stated in [OMAN 89] the more clues a novice can get the better they can debug a program. If JSP is suitable for both the novice **and** the more experienced designer then this can hopefully reduce the emphasis experience has on debugging.
- Teaching the novice becomes easier. The novice can learn the JSP constructs without the need to the learn the syntax of the language's control constructs. The onus of a JSP COBOL programming course could then be design, rather than expecting the student to learn both techniques simultaneously.

This debugging tool would hopefully increase the productivity of a novice, but **is it of any use to an experienced designer?** The answer, hopefully, is an emphatic, *YES.*

- The experienced designer faces the same impulse to 'fix' the source that a novice does (although they should know better). So this removes all temptation to look at the code.
- The experienced designer faces exactly the same debugging problems as a novice, and the more clues available , the faster programs can be debugged.

As to the actual process of JSP development.

- It would hopefully be done faster, because less time is spent at the code level:- fixing, debugging, testing and running.

- It would hopefully be more enjoyable; as rooting about in the code for elusive errors can be frustrating, and often unproductive.
- It places the emphasis of JSP development back on design. Too often the JSP design process is used as a guideline for producing code. The JSP design should **be** the code, and this helps the developer to view it as such.

# Part II
# Approach

## 10 Compiler Design

Most formal compiler texts present little in the way of design to be used when implementing a compiler. Formal notation such as Backus Naur Form is used to describe the syntax of the language, but the semantics of the language and the implementation of those semantics are usually described informally [ROSSELET 87].

Algorithms that can be used in the construction of a compiler are freely given [AHO 86], [BORNAT 79], and in doing so these texts describe the modular nature of compilers and compiler passes.

Other texts supply the information needed to build compilers [MAK 91], [HORWOOD 90], [HOLUB 90]. Providing source code for the algorithms and how they interact, but no time is spent explaining how they come to fit together in this way.

Compiler design would appear to be picking the best algorithms for the job and stringing them together to form a whole.

In an attempt to understand how compilers & interpreters work and to try and attach a coherent design methodology onto an otherwise adhoc development, JSP has been used to design the most procedural parts of the interpreter.

### 10.1 JSP, a compiler design methodology

The main tasks of an interpreter can be summarised as :- build a symbol table, then use this to parse the control structures and statements of the source language.

The information to be processed by the current project was conveniently split into the following:

**A '.ops' file** which contains all the operations in the current program in the following format.

```
OP_NUMBER  STATEMENT
           STATEMENT
OP_NUMBER  STATEMENT
```

e.g

```
001 MOVE 2 TO COUNT.
    ADD 12 TO COUNT.
002 MOVE "DONE" TO MESSAGE.
```

**A '.cnd' file** which contains all the conditions in the designed program. These are described by a similar format.

```
COND_NUMBER CONDITION
```

e.g

```
C01 LOOP > 10
```

**A '.dat' file** which contains the COBOL descriptions of all identifiers and files. It consists of the usual COBOL divisions.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
```

The current project was interested solely in the DATA DIVISION, and more specifically the WORKING-STORAGE SECTION within it.

**A '.pr0' file** which contains the JSP diagram. This is the program's graphical representation containing all the allocated operations and conditions used.

**A '.qui' file** which contains all the quits used by the backtracking constructs, this was not used in the project.

The process of JSP was used to examine each of the files, and use each one of these to produce an aspect of the interpreter. The '.dat' would be used to generate the symbol table, the '.cnd' and '.ops' would then be parsed and cross referenced with the symbol table to produce tables of conditions and operations. The '.pr0' file would then be used to execute each of these operations and conditions as and when they appeared on the diagram.

The interpreter design then became a normal JSP design problem treating each of the files as the input data, and using these to determine the procedural aspects of the process. The aspects not explained in compiler texts.

### 10.2 Mixing Data Types and Data Structures

The manipulation of the data structures such as the symbol table, could not be done using JSP and would not want to be done through the use of JSP. As exemplified by [SANDEN 89], JSP would result in *"an impractical design with unnecessary tasks"*. So these were written using the techniques presented in the texts, as data types. All functions which act upon these types are atomic and effectively become language directives.

This 'mixture' of techniques is consistent with Jackson's own view of design;

> *The program we are trying to build consists essentially of operations to be executed by the computer. Some of these operations are concerned with finding our way around the data structures; when we are ready to pass on to the next item we must execute the operation "read item file" ... Other operations are more directly concerned with the task to be performed: for each item in an invoice we must execute the operation "add price to invoice total"*[JACKSON 75, pp.11]

Essentially this means that Jackson recognises two types of operations, low level operations not strictly modelled by the data structure such as file operations, and operations that deal with the data structure such as the invoice `add to total` operation.

All functions acting upon the interpreter's data structures such as the symbol table or keyword trie become low level operations. This avoids any potential structure clash between these structures and the input file structures. It also means that 'C' elegance can be used to provide recursive, and compact solutions to the problems of manipulating these structures.

## 11 JSP and 'C'

The chosen implementation language for the project was 'C'. Compiler & interpreter construction is very often done using 'C'. And the JSP approach

seems natural when the view taken of the parsing procedure is that of processing input files. A problem occurs when combining the two techniques.

There is no difficulty using JSP, but when the project was started there was no means available to generate 'C' code. This led to the decision, initially at least, to hand code the diagrams. This was to be done with as much flexibility for future inversion as possible, and led to unmanageable code being produced.

The reason being that there were a number of possible ways of implementing the inversion technique in 'C', and the most flexible was the one that was the least natural to code.

## 11.1 Inversion in 'C'

The inverted functions are not sub-routines of one another since they have internal states which are independent of the other routines in operation. The term 'co-routines' is used to describe these routines. So when a function passes control to another routine and then control returns to the first routine, it must start processing from the point that it relinquished control.

The solution could be approached in a number of ways.

1. treat each sub-program as a state machine and process it as such

e.g.

```
switch(state)
        {
        case state1: first();
        case state2: second();
        case state3: third();
        }
```

Unfortunately this solution removes any JSP design from the program.

2. Treat it as a state machine but retain the JSP structure

e.g.

```
if state1 then goto after-state1
else if state2 then goto after-state2
     else ......
```

The if statement restarts the program from the point immediately after the return invocation, the rest of the JSP generated code remains unchanged. This does mean that normal 'C' coding techniques such as nested if's, while loops, and modular function calls must be avoided since it is not possible to jump into the middle of any of these constructs.

3. Program the code with the knowledge of the inversion process so that each segment of the code before the return is a function which executes to completion and the point immediately after the function is a return invocation. When the program restarts it jumps to the point immediately after the return and this happens to be a function call.

   This solution is basically the same as the solution presented above but it removes many of the goto's and puts functions in their place.

```
        e.g
                    before_return();
                    return();
after_return:       function_after_return();
```

   The above solution is worse than any of the previous two since it is not as easy to modify - assuming that a return becomes necessary within a function, and then that function has to be split into two with a return between them.

   It is also not so obviously derived from the design since the above code segment could be derived from one box on the diagram.

   The solution also allows a fear of goto's to dictate coding in such a way that it does not follow an otherwise structured design, even although it results in code with no structured constructs.

### 11.1.1 Inversion in the Interpreter

Solution 2 was chosen as the most mechanical but for large programs it became very difficult to hand code effectively. This resulted in the construction of a program which could take the code generated by a JSP diagram and convert it into compilable 'C', this is explained fully on page 90.

This mechanical approach to 'C' coding unfortunately meant that no time could be spent working out a method to invert the programs correctly.

Leading to intermediate files being used as the communication method between all the interpreter's modules.

This meant that routines had to be written to save all the tables that were built up by the scanning and parsing routines i.e. the symbol table, the table of operations and the table of conditions. This introduced an added time overhead, but also suggested ideas for further development of the project. Opening up the whole area of persistent data, and incremental interpretation. This is suggested by [FRITZSON 83] who used a persistent symbol table to develop a symbolic debugger using incremental compilation.

# 12 Diagrammer Design

Initially the project consisted of the development of a jsp diagrammer, with the diagrammer having interactive debugging facilities. A project of this scale in the time available did not seem to be feasible.

The project was re-specified so that PROBENCH files would be used. However, a large amount of groundwork had already been done on the diagrammer. This work had resulted in a simple diagrammer that allowed the creation of a tree structure containing boxes. These could be deleted, viewed, and added in numerous ways,.

As soon as it became clear that further work on the diagrammer would be detrimental to the production of the interpreter, all work on the diagrammer ceased.

Design of the diagrammer was done using a mixture of Algebraic Specification and case analysis.

A number of low level data structures were identified, and operations to act upon them were defined. The most rigorously specified structure was the boxlist (section C.6.2 & section G.2.2), this is the main structure used in the tree representation. Most of the other data structures had simple operations such as create or display, which were designed very simply.

# 13 Development of the Interpreter

As mentioned in section 10.1 The interpreter is concerned with 4 of the files used by PROBENCH.

1. '.dat' used to prepare the symbol table
2. '.ops' used to generate the operations table

3. '.cnd' used to generate the conditions table

4. '.pr0' which contains the JSP structure

The interpreter's programs are connected by intermediate files, these are either built up from tokens, read or written by the code on page 409. Or they are the saved versions of already built up data structures such as the symbol table, operations table, or conditions table.

## 13.1 Preparing the Symbol Table

The '.dat' file is the one file PROBENCH uses, that fully adheres to COBOL syntax. This is because PROBENCH does no processing on this file, and simply tags it on the front of the generated code.

The file contains all the definitions for the COBOL program, all variable declarations, all file definitions, and the program information sections.

The file follows COBOL's strict column syntax. Based on punched cards, the first six columns are used for page and line numbering. Column 7 contains the line type. Columns 8 through 72 contain all the coding information, and columns 73 through 80 are known as the identification area and are reserved.

The indicator area, column 7, determines the type of the line. A comment line is indicated by placing an "*" in this column. A print line, i.e. means that this line is placed at the top of a new page when printed, is represented by "/". A continuation line is represented by a "-", this means that the line above has been continued exactly where it left off. If a non-numeric literal has been continued then the continuation begins with a ' on or after column 12.

An analysis of this file resulted in a 3 separate data structures being produced. The first considered the input file ('.dat') as a number of lines each of different type, with the lines consisting of columns. The second considered the input file as a number of lines of statements. The third as a number of tokens.

### 13.1.1 The Line Scanner

The first view of the input file produced the program contained in section F.1 on page 181.

This program takes the '.dat' file and produces as output a file with each line on the file containing a separate program statement. To do this

it removes columns 1 through 6. Any comments are ignored. Continuation lines are continued on the same line with any extra spacing removed, unless the item being continued was a non-numeric literal, in which case the continuing ' is removed and everything after this is written to the output file. Page breaks are ignored with the line simply being written to the output file. Columns 73 through 80 are ignored.

A statement ends when the last character on the line is a fullstop.

Although no consideration has been given to syntax errors, it is at this stage that errors relating to the structure of the records would be handled.

### 13.1.2 The Scanner

With the regimented line structure of COBOL removed, it becomes possible to regard a COBOL statement as a sequence of characters. Or more specifically as a number of statement parts, each part possibly having a number of spaces, followed by a recognised tokenisable part (the program can be found in section F.2 on page 209). The recognised tokens are non numeric literals delimited by " & ", or ' & '. Open parenthesis and close parenthesis '(', and ')'. Fullstops and keywords or non-keywords.

Processing of a non-numeric literal consists of skipping the start delimiter, and processing all characters until the end delimiter is reached, at which point it too is skipped. A token is written out of type NON_NUMERIC_LITERAL, which contains the length and contents of the string. It is possible to have a string delimited by ' characters to contain these characters, this is done by placing two of them next to each other. Therefore the following string 'al"s' is actually al's. This is catered for in the program by using one character read ahead.

'(','(' and '.' cause a special token to be written to the output file.

In order to distinguish keywords from other words in the text it was necessary to store the keywords in some easily accessible data structure. The chosen structure was a trie (explained fully in section C.5, page 79), which was perfect for the type of processing done in this program. A trie allows access to its data by checking on character at a time and this is exactly how the characters from the file are processed. The main problem being that this involves a large amount of storage space to contain the structure. Too large, unfortunately, for all the PC's it was tested on. As a result, the interpreter modules only work on an ATARI ST with 1024 K of memory available.

Anything not recognised as a keyword is written out as a sequence of characters preceded by their length.

### 13.1.3 The Symbol Table Generator

This program (section F.3 on page 233) takes the tokenised output from the scanner and uses it to build up a symbol table.

Everything before the WORKING-STORAGE SECTION is skipped, in the current implementation we are interested only in the variables used by the program.

The very first 01 level declared in the program is stored separately, the intention being that all symbol tables nodes could be displayed in the order in which they occur in the '.dat' file. Although this display mechanism was not implemented.

Two low level data types are used in this program, one being the symbol table itself (discussed on page 75), the other being a stack (page 78).

The stack is used due to the absence of recursion in JSP. Each level has a lower level parent ie an 02 level has an 01 parent. Each 02 level may be the parent of a higher 03 or 04 level child. In order to keep track of this relationship a stack was used. With each new level encountered being pushed on the stack, popped by child processes to find out who their parents are and eventually removed when a new level of the same number comes along. This allows a recursive solution to the parent child referencing problem to be implemented in a non-recursive design.

It is important that these relationships be maintained. Not so much for the current implementation. But for future enhanced versions where constructs such as MOVE CORRESPONDING will become available, and all the corresponding children of two identifiers will have their values moved to children of another variable.

An identifier is expected to take the form;

```
LEVEL_NUMBER IDENTIFIER_NAME FULLSTOP
```

or

```
LEVEL_NUMBER IDENTIFIER_NAME PICTURE FORMAT FULLSTOP
```

The level_number is used to determine the parent and subsequent siblings. The identifier_name is hashed to determine the symbol table slot that it will occupy, each identifier is also given a unique number because the same name may exist for a number of identifiers. The format is parsed but used

in a limited way, if the format consists of 9's, i.e. an integer variable, then all that is noted is that the variable is an integer and is represented using one. For PIC X fields the number of X's are counted and this much space is allocated for the character strings.

Identifiers are added to the symbol table in the form of nodes when they have been completely parsed.

This program saves the symbol table for use by the other programs which make up the interpreter.

## 13.2 The Operations Table

The '.ops' file contains all the operations and their associated numbers as used in the JSP design. This file is simpler to scan than the '.dat' because there are no columns to consider. The file is processed character by character so there is no concept of records, simply operations.

The file consists of a number of operations, each beginning with a 3 digit number which identifies it, and a number of statements which make up the operation. Each statement ends in a fullstop, this is a restricted form of COBOL syntax derived from COBOL '74.

This method of identifying the end of a statement should be changed in future versions, but was necessary in the current because no distinction was made between keywords. The design assumption made was that all keywords are equal, in fact they are not. Some keywords can only be used at the start of a statement, these are all the verbs such as MOVE, and DISPLAY. Others such as IS, and TO can only be used in the context of another keyword. Had this division of keywords been used then there would be no need to enforce the fullstop convention.

### 13.2.1 Scanning the Operations

The first program (section F.4, page 260) takes the '.ops' file and produces a file containing all the operations in token form.

The file is scanned, taking the operation number as 3 numeric characters and converting it into an integer value.

The operation is read character by character; skipping blank spaces; converting non-numeric literals, fullstops, keywords, identifiers and numeric literals into tokens. This is done until the end of the line is reached.

Once a line break is encountered, the next step is to determine if the next line is a new operation or a continuation. The processing continues

until the end of the file is reached.

### 13.2.2 Parsing the Operations

Parsing the operations (section F.5, page 287) is a relatively simple process. The operation number is read from the file, this determines which slot in the operations array to use, this is a fixed size of 999.

Each slot contains a linked list of statements. Each statement consists of a linked list of tokens, with the full stop token becoming a NULL link.

The tokens are added to the statement list until a fullstop is encountered. The next step is to either begin another statement list or to start building up a new operation slot.

Once the table has been built it is saved to a file for future use.

This process is simple because no error checking is done, and a fixed form of operation has been used.

In order to evaluate operations there were two choices available.

1. Write JSP code which evaluated operations.

2. Assume that an operation knows how to evaluate itself and have the evaluate_operation function become a low level directive which can act on the data structure.

Solution 2 was chosen as this was the simplest and removes any structure clashes from future JSP programs which need to evaluate operations.

## 13.3 The Conditions Table

The '.cnd' file processing is done in a two stage manner similar to the processing of the '.ops' file.

The same scanning method is used to handle this file as was used on the operations file, meaning that the file is read one character at a time. Again removing the view of a file containing records, instead the file contains a number of conditions.

A condition consists of a condition number, the format of this is a two digit number preceded by the letter 'C', valid condition numbers including C01, C11, and C99.

The rest of the condition then consists of identifiers, literals, and relational statements.

### 13.3.1 Scanning the Conditions

The first program processing conditions is the condition scanner (section F.6, page 299). This takes the '.cnd' file character by character picking out the condition number, and then each of the parts of the condition.

The condition number is read in as two numeric characters, these are converted into an integer value and written out to the intermediate file.

The rest of the condition is read until either the end of the file is reached or another condition is found. Valid items in the rest of the condition are non-numeric literals, relational symbols, numeric literals, fullstops, identifier names, and keywords.

Valid relational symbols are '>', '<', and '='.

The component parts are all written out to the intermediate file in the form of tokens as before.

### 13.3.2 Parsing the Conditions

The conditions that the Interpreter will parse have been limited to the following format.

```
LEFT-HAND-SIDE   < NOT > RELATIONAL-OPERATOR    RIGHT-HAND-SIDE
```

The NOT is optional, and the following comparisons are available.

- <,>, and = can be used with numeric literals, and numeric variables.
- non-numeric literal and character variable comparisons can be carried out with the = operator.

The parsing is carried out as follows (section F.7, page 329):

1. read the condition number
2. read the left-hand side
3. read the relational operator
4. read the right hand side
5. then move onto the next condition or end

The condition is then stored in the condition array, this is a fixed size of 99 slots. These are then saved to the condition-array file.

A design decision was taken to have the code which could evaluate a condition be an action on a condition data structure. This means that instead of having a JSP program which steps through a diagram and attempts to execute a condition using JSP, the program simply calls the evaluate_condition function.

As with all other data structures this removes any structure clashes from the design method.

## 13.4 The PROBENCH Diagram File

The design of the '.pr0' file is contained in section G.3. The file contains the boxes in box_number order.

In order to manipulate the diagram outwith probench a tree has to be built up using the details on the file. This is done through the use of the diagrammer functions.

The first box is read in and made the root box. All subsequent boxes are added under the parent box. The number of the parent box is found by taking the go-back number of the current box, and subtracting this number from the current box number.

Due to certain differences between the ATARI-ST and the PC, the file can not be read directly on the ATARI-ST. I suspect that the main difference is the word length used by the computers. The ST uses a 32 bit word for integers and addresses while the PC uses a 16 bit word. This means that when an fwrite or fread is carried out the subsequent memory representations of integers do not correspond.

With a little more time this problem could have been resolved by translating the file into a more amenable representation. This would have taken a form similar to the intermediate files used by the interpreter where integers are written out using the fprintf("%d") directive.

## 13.5 Development of the Diagrammer

Algebraic Specification and case analysis were used to design the low level data structures and functions which make up the diagrammer. These low level functions were nested within higher level constructs such as 'Add_box_under' & 'Delete_box'.

A high level parse routine was written which called the high level functions to manipulate the diagram. The parser was built up incrementally and although it was only intended to be a test function it expanded to such an extent that re-writing it would have taken too much time. The need for such a re-write only became apparent after development of the diagrammer had ceased.

The on-line trace for the interpreter was intended to be done via hooks into probench to highlight boxes as the run progressed. This was in itself a large body of work which could not be carried out in the time remaining. Once this was realised, the diagrammer again became a major part of the project.

Routines were written to load a PROBENCH file into the diagrammer but this only worked on the PC, and the interpreter modules only worked on the ST.

Time was growing short and in order to actually achieve on-line interpretation of a JSP diagram there were two solutions available.

1. Write a program which could read the probench file and save it in a generic form that could be swapped between the two machines.

2. Ignore the '.pr0' file totally and expand the diagrammer to allow sequence, selection and iteration constructs.

Solution 1 could only be feasible if it worked first time, there would be very little time to fix these routines if they didn't.

Because of the uncertainty of solution 1. Solution 2 was chosen. At first glance this appears to be the most complex solution, but the probench representation of a box had been combined with the diagrammer representation of a box when the load function was being written. This meant that boxes already had a type field, and condition field so routines just had to be written to alter these.

The resulting routine allowed boxes to be declared as type sequence, selection or iteration. Conditions could be allocated to boxes, whose parents were selection or iteration boxes, and operations could be allocated to boxes with no children.

Having done this, the routines which single stepped around a diagram could be implemented within the diagrammer and the interpreter could be used to create and debug diagrams.

# 14 Testing

## 14.1 The Interpreter

The parts of the interpreter were written in the following order:

1. Processing the Symbol Table
2. Processing the Operations
3. Processing the conditions

### 14.1.1 Testing the Symbol Table

The symbol table is built up from the '.dat' file so it was important to have a '.dat' file that worked. This was ensured by using an old COBOL assignment and cutting down the '.dat' file. The '.dat' file was artificially expanded to test for comment removal, continuation line handling and print line handling.

The first program was written and tested on the code. And the output file was examined to ensure that the correct actions had been taken. The second program was then written.

Using the output from the first program, the second program was tested using this as input. The output from this program was in a form difficult to understand, as shown below.

```
4
145
4
86
3
3
4
225
3
3
6
8
5
VALIDATE
3
3
```

The comprehension of the file was made easier by using 'fprintf' rather than 'fwrite', this allowed newlines to be taken after each data item, and all the integers are represented using textual characters.

This file was checked to ensure that everything was correct and the third program was written.

This program was tested by using the output from the previous program as input, displaying the symbol table on screen, then saving the symbol table.

An extra program was then written which used the *'load_symbol_table'* command to load the symbol table, it was then printed out and saved.

The three symbol table programs communicate via intermediate files.

### 14.1.2 Testing the Operations

The Operations programs were tested in much the same way as the symbol table programs. A file was created containing operations acting upon the identifiers in the '.dat' file and this was run through the first program.

The output, which was in tokenised form was examined, and then the second program was written. This took the output from the first, built up the table of operations, printed it, and saved it to disk.

Again an extra program was written which used the function to load the operations, these were printed and saved. The main difference with this program was that the operations could be evaluated.

### 14.1.3 Testing the Conditions

The was exactly the same as the above testing procedure with the output of one program being examined and used as the input to the next program.

Again the conditions could be evaluated in the last program.

## 14.2 The Diagrammer

The diagrammer was actually written and tested before the interpreter routines. It is being discussed after the interpreter section because the diagrammer was expanded to use the above tables and routines.

The diagrammer was easier to test than the interpreter modules because it is interactive, it could be used and any mistakes would be discovered quickly.

The diagrammer user guide on page 70 lists all the operations available within the diagrammer. The addition operations were tested: add_under, add_above, add_left, add_right, and all the multiple add operations.

The diagram was displayed both graphically and textually. Shortcomings were found with the display routines because they take no account of the size of the screen, so for large diagrams in textual form the screen scrolls up automatically losing the first displayed boxes. And graphically no windows are used so the diagram is not constrained by screen edges and wraps around. With a little more development time both of these could be solved quite easily.

The multiple box additions were tested to some extreme degrees with boxes being given several thousand children. The diagrammer took some time to add them, the difficulty occurred when attempting to display them. The display function is recursive and it therefore uses the internal stack, the 'C' compiler does not reserve enough space for the display operation to use and so the program runs out of stack space. This is a display problem not an addition problem, because the total number of boxes was increased to the correct number after the addition.

This testing was all carried out before the interpreter routines were added, once this was done a number of test diagrams were created.

## 14.3 Testing the Debugger

Two test programs were created, these were run through the interpreter routines to get the tables created and then these were loaded into the diagrammer.

With only three operations available it was difficult to come up with suitable test programs, but the following programs test various aspects of the diagrammer and interpreter.

### 14.3.1 Multiplication

The first program, section D.1 was designed to carry out a number of multiplication calculations using the following output.

```
 1
*
 2
---
 2
```

---

The numbers used in the sum were to increase as follows, the number to be multiplied would increase by one, and the multiplication factor would increase by 2.

The program would stop when the top number reached 11.

In order to use the program the diagram had to be created on the diagrammer. The operations, and conditions were added and then single stepping of the program began.

At first the answers were not what was expected, but by single stepping again the bug was found to be no occurrence of operation 1, this resets the answer to 0 between calculations.

This program tested the single stepping of the debugger, and iterations.

## 14.4 Textual Loops

The second program was to produce the following output.

```
ONE
TWO
TWO
THREE
THREE
THREE
FOUR
FOUR
FOUR
FOUR
FIVE
FIVE
FIVE
FIVE
FIVE
```

The program design is shown in section D.2, the program was run to completion and worked first time. It tested iterations, selections and the run to completion feature.

# Part III
# Evaluation

## 15 Evaluation

The evaluation of the project will first begin with the deliverables. The main products being the COBOL interpreter and the diagrammer, although the spin-off code translator has also become a large part of the project.

The project was not only concerned with the development of a large software package, but also with the application of techniques not normally associated with compiler development. Some time will be spent evaluating the effectiveness of these techniques.

### 15.1 The Deliverables

#### 15.1.1 The Interpreter & The Diagrammer

The COBOL interpreter will tokenise all COBOL verbs, although no syntax checking is carried out. Only three verbs can be executed, ADD, DISPLAY, and MOVE, each with a constrained syntax. The available conditions have been limited to simple comparisons but this does allow simple iteration and selection programs to be written. The working storage section is the most limited aspect of the interpreter with handling only of integer and character variables.

Although the interpreter is not particularly functional, the reasons for building it were *not*, to simply produce a COBOL interpreter, but to examine the effects of combining an interpreter with a JSP diagrammer.

The diagrammer which has been developed allows the addition and deletion of boxes, the allocation of operations and conditions, and the displaying and debugging of a diagram created within it.

The debugging aspect of the project involved combining the interpreter with the diagrammer. This caused some problems (see section16), but once combined it allowed diagrams created using the diagrammer to be allocated operations and conditions prepared by the interpreter. The diagram could then be executed without leaving the diagrammer.

Obviously this is a slightly artificial situation, because all the operations and conditions have to be prepared outwith the diagrammer. But in a fully functional implementation, the diagrammer would allow the creation and

allocation of both operations and conditions, these could then be prepared on-line and the diagrammer debugged interactively. This is the main reason Probench files were used in the project, the intention being to interface with this product to allow the on-line debugging. Unfortunately no interfacing between the two products occurred.

It should perhaps be mentioned that the products have not been developed to the stage where they could productively be used by anyone other than the developer. But the development *and* use of the products, has provided an insight into the implications of larger and more substantial developments of this type.

### 15.1.2 The Code Translator

Developed as a necessary side effect from choosing JSP as the design methodology. The Code translator takes 'C' programs written in probench, and generated using the COBOL GENERATION option, and translates the output into code which can be compiled using a 'C' compiler. It allows the use of sequence, selection, iteration, backtracking and subtrees although no support has been provided for inversion.

The use of the translator allows the quick production of programs, however when debugging these programs the programmer experiences the problems outlined in section9.2. Especially the continual movement from diagrammer to compiler. This is made even more intrusive by the need to run the translator on the source file before compiling.

This could not be helped, and is a problem faced by most JSP developers using this technology. But the debugging process could be made easier by removing all redundant code from program (see section 17.2.5).

## 15.2 The Approach

The approach taken towards design in this project was mentioned in part II. The reasons for using JSP as a design methodology when designing compilers were discussed. The effectiveness of this approach will now be discussed.

### 15.2.1 JSP a Compiler Design Methodology

JSP was chosen as the design behind the interpreter because parsing is basically an input/output process with some processing being done to the input data. This is what JSP has been used for since its conception. As a result it seemed reasonable to be used in interpreter design.

The solution to the parsing problem was produced, it was a little larger than the comparable 'C' routines, but this was a stand alone program rather than a routine. It did the job, perfectly.

As to tokenising, JSP again produced an acceptable solution. So JSP as a design methodology is fine, however, each of the scanning and parsing routines were stand alone programs.

There was no method available to invert the separate routines together.This is not a fault of the methodology, because when the programs were hand-coded from JSP, inversion was possible. The problem lies with the tools used to carry out the design.

Had the interpreter been written using COBOL then there would have been no problem producing an integrated system, tools exist to invert COBOL JSP diagrams. No such tool exists for the 'C' programmer.

So while the design is not problematic, the actual code production *is*. There are two ways to proceed from here. One way would be to use JSP as a design method but then hand-code the solution using the JSP design to influence the program. The second is to hand-code the solution, but to use the design to direct the program.

The second method was attempted, but the chosen conversion method was too difficult for large programs. The first method is unsuitable. To use a design method simply to influence the coding means that the design may not be done fully. It may be half done, just to get an idea of the program, and then coding will begin.

Of course the design may be carried out completely. But then a change needs to be made. The change can be made to the design quickly and easily on the diagrammer but the change to the code may not be quite so easy to implement. This could either be because the change was so fundamental, or because the coding technique used did not allow for this type of change.

The repercusions this has on using JSP to design compilers are as follows.

- JSP should only be used when the code can be generated.
- JSP should only be used if the designs can be inverted together.

Having said this the process of understanding parsing and scanning can be aided through the use of JSP. This is because solutions have to be complete so the finished design completely reflects all the mechanisms used. And the design must be fully understood before coding can begin, this means that the scanner can not be written until the technique is fully understood.

So for the first time compiler writer, the process of design in JSP can highlight subtle difficulties and techniques. They are then more fully prepared to construct the same product using a different design methodology.

It is unfortunate that 'C' cannot be produced using the available JSP diagrammers because the technique lends itself perfectly to the construction of parsers, scanners and syntax analysers.

### 15.2.2 Mixing Data Types & Data Structures

Section 10.2 discussed how low level functions were used to act on data types, and these could then be used as operations on the JSP data structures.

This approach is a mixture of algebraic specification and JSP, although very little actual algebraic specification was used in the production of the product, the technique of case analysis was used to produce data types and functions acting upon them.

This approach has the benefit of using JSP to provide the structure of the program, and using functions to do most of the actual work. This should really have been used to produce the interface to the diagrammer, i.e. the job that the parse() function now does, but the parse function was built up with the intention of testing the diagrammer functions rather than driving them.

## 16 Problems & Setbacks

Most of the main stumbling blocks encountered during the course of the project should have been mentioned before, but they will be summarised here.

1. There were no tools available for the generation of 'C' code from JSP. This led to some time being wasted attempting to hand code the designs, and time being spent developing a utility to produce 'C' code.

2. No inversion was possible once the generated 'C' code was being used to implement the project. This led to intermediate files being used. These often took a substantial amount of development and testing time to complete.

3. The memory intensive data structure used to hold the keywords prevented the interpreter modules from being run on the target machine.

4. File incompatibility between the development machine and the target machine meant that the completed diagrammer could not use the diagrams created by probench, only the text files used to drive the diagrammer.

## 16.1 Problem 1: Unproductive Development Time

The time wasted on problem one could have been avoided by using the diagrams as guidelines rather than absolutes. This would have allowed more sophisticated 'C' constructs to be used, and the code produced being more elegant.

The danger with this approach would have been the writing of code using a loosely defined diagram which did not take into account the full scope of the problem. Thereby resulting in extreme changes which effected a re-write of much of the code.

I hold the opinion that if a design method is being used then the design should direct the code rather than inspire it. This being the main reason for choosing the mechanical coding approach.

The time lost could have been minimised. Had it been realised sooner that some code would be extremely difficult to produce. This would have resulted in the Code translation utility being started earlier and allowing more time to be spent on the production of the debugger. This was due to an ineffective feasibility study.

No matter what the reason, the approach did result in a utility to generate 'C' code. This can now be used by any 'C' programmer to apply JSP techniques to the production of code.

## 16.2 Problem 2 : Lack of Inversion

The 'C' generation utility did not support inversion. This led to all communication between programs being done via intermediate files.

These files took time to design and test. Some were quite difficult to design because they involved the writing to disk of dynamic structures that were linked via pointer chains.

This again meant that time was not spent on the development of the debugger. However once done, these routines could be used in a more fully developed system to increase development speed.

Incremental Compilation could be used to interpret the operations, conditions, and symbol table as they are entered, these could then be saved

along with the '.ops','.dat', and '.cnd' files. When the diagram was reloaded the data structure files could also be loaded, preventing the user from having to wait until the '.ops','.dat', and '.cnd' are re-interpreted before recommencing development.

If the system were fully developed then it opens up the notion of persistence to the developed programs. It may not be the case that the program has to start from the beginning when a run has finished. The program could be held in an unfinished state between executions, by saving the symbol table and trace information, and the program restarted from this point.

### 16.3 Problem 3 : Memory Intensive Data Structures

The trie used to store keywords is the most applicable data structure for the problem, it is unfortunate that it consumes so much of the computers memory. Section 17.2.1 provides some solutions to this problem which were not implemented due to a lack of time.

### 16.4 Problem 4 : File Incompatibility

Again this problem could have been resolved had there been enough time left to attempt the solutions. However, the incompatibility did not prevent the debugger from being built, and the investigation into its benefits was carried out after hands-on experience.

## 17 Future Work

### 17.1 Enhancements

Enhancements are considered to be changes that require no design re-development in order to implement, the project as it stands is open to few enhancements.

The syntax of the available COBOL constructs could be expanded;

- The verbs, ADD,DISPLAY and MOVE, could all be expanded to allow a fuller COBOL syntax.
  - With a little work, DISPLAY could take more than one argument and the NO ADVANCING clause could be added. This could be done quite simply by enhancing the DISPLAY verb implementation to loop through all of the arguments until it either came across a NULL link when it would stop. Or until the WITH

keyword was found, at which point the function would return without printing out a newline symbol. This takes no account of the UPON clause which would require more work to be done on the '.dat' processing.

- The MOVE verb could be expanded quite easily to move both numerics and strings. The string MOVE would have to take into account the character variable's length but this is already stored so wouldn't be too much of a problem.

  Move could also be made fuller so that the following syntax applied:

  ```
  MOVE identifier1 TO identifier2 ...
       literal-1
  ```

  This allows one value to be moved to multiple destinations. This would be done in a similar fashion to the display by looping through all the destinations after the TO keyword and carrying out a move.

  This does not take into account the CORRESPONDING construct which should be possible to implement, using the current structures, but would require more work. It would need to be made to work through all the lower levels of an identifier, matching up variables name.

- ADD could be expanded to add a number of variables to a number of other variables.

  ```
  ADD identifier1 ... TO identifier2 ...
      literal-1
  ```

  By totalling up the values of all the numerics to the left of the TO keyword, this could then be added to each of the identifiers on the right. To take advantage of the ROUNDING or SIZE ERROR constructs, the '.dat' processing would have to be enhanced to make more intelligent use of the format fields. Rather than storing a PIC 9 field as an integer, it should be stored as character, with the integer value of the character string being converted before and after every use.

The interpreter in the diagrammer could be enhanced by displaying the Conditions which are evaluated as the trace around the diagram progresses.

This would require a change to the interpreter routines so that every time a condition is about to be evaluated it is displayed on screen.

There are a few HCI changes that could be made to the interpreter. The textual display could be altered so that it pauses every time 20 boxes are printed, this would allow the user to control the scrolling of the text on screen.

A newline before every prompt could improve the screen display. As would user positioning of the diagram's graphical display, this can often be obscured by text.

Most of these changes are very small and cosmetic and would have been implemented had the diagrammer not been halted at such an early state of development. However if the development had not stopped then it is unlikely that the interpreter would have been developed to its current position.

## 17.2 Future Developments

### 17.2.1 Keyword Recognition

The interpreter routines could be changed to work on the PC. The only thing that requires altering in order to do this, is the manner in which keywords are distinguished. At the moment a trie is used, this could either be altered so that it resides on disk rather than memory. Alternatively a different approach could be taken, the recommendation being a series of arrays, each holding keywords of different length. This could cut down the time needed to find a particular keyword because only the array holding the required length would be searched sequentially. Each array entry would need to be a tuple containing the keyword name as a string, but also its token code.

The changes needed to implement this would be to the parsing mechanism , so that a full word is read in before it is checked, rather than one character at a time. But it would also do away with the list of keyword strings, used when displaying the keywords in operations, because this is now stored in the array with its token code.

### 17.2.2 Hooks into Probench

Once the above change has been made it should be possible to run the interpreter on a PC. An interface between the interpreter and Probench could be written so that the interpreter displayed the trace graphically on the diagram rather than in text form. So instead of displaying BOX 1, and

OP_NUM 2. The box and the operation would then be shown in inverse on the diagram.

### 17.2.3 Symbol Table Handling

The symbol table could be expanded by storing not only the current value of each variable, but also the initial value of the variable. This would allow the use of the VALUE clause in the '.dat' file, and would enable the interpreter to restore all the variables to their correct state when beginning a debugging session. This would need to be done before any serious expansion of the project could be considered.

### 17.2.4 A Fully Functional JSP Debugger

There are obviously far more changes needed than the few mentioned before the Interpreter and Debugger could become fully functional. It would be folly to list them all here since the list would include a description of every COBOL feature and every feature available in a good debugger.

However, one of the project aims was to determine the feasibility of developing such a product. And with the progress made in this project it would seem to be perfectly feasible, given a slightly different approach and a larger time-scale.

### 17.2.5 Removal of Redundant Code

The code produced by Probench contains a lot of redundant information, for example GOTO's are issued which send control to a label whose only processing is to issue another GOTO.

These could be removed by a program which scans the source file in two passes. The first time to analyse all the labels, and link any labels which are connected via immediate GOTO's. The second pass could then check each GOTO, and if it was sending control to a label which has a follow on link then this link could be pursued until no follow on link was found , the GOTO in the code would then be replaced by a GOTO to this end label.

Obviously more processing would be required to remove the now unused code, the intermediate jumps.

Essentially this process is quite simple, the hardest part being the building up and maintaining of the linked lists of labels.

This algorithm was not implemented during the current project because the time it would have taken to implement would not have outweighed the time saved by its development.

Until a debugger is available for source level debugging of the diagram it will always be a problem to debug this type of code, so there does exist a need for a program of this sort.

### 17.2.6 Enhancements to the Diagrammer

The future developments listed above are all related to the interpreter, this is not intended to suggest that the diagrammer does not need further development. In order to be useable by anyone, other than myself, it most definitely does need to be developed further.

The developments necessary are not mentioned because the JSP diagrammer needs of the developer are adequetly met by a number of programs, i.e. Probench.

The current diagrammer, when expanded, would offer very few advantages over these programs. The only obvious advantage would be fewer constraints on diagram size, since the only constraint on the project's diagrammer is memory. These limits do not seem to have affected the usefulness of the available products, and so the appeal of the enhaced diagrammer is likely to be limited.

## 18 In Conclusion

Although JSP is an old technique, the tools to support it are continually being expanded. It seems natural to expect that some attempt in the future will be made to combine interpreters and an interactive debugging environment with these tools.

There are dangers with this approach, any tool which expands its range of features, often does so at the expense of flexibility.

By implementing a COBOL interpreter useable by Probench, Probench immediately becomes useful only for producing the COBOL code that the interpreter can handle. It can no longer be used to produce code which will use MF-COBOL screen handling routines. Rather it can, but the users of these routines will not be able to test this part of their project within the interpreter, unless the interpreter was made compatible with this product.

This implies that the COBOL interpreter would have to do much more than simply interpret all valid COBOL instructions, it would also have to

be compatible with a number of vendor specific COBOL constructs. That is, *if* Probench is to remain as flexible as it currently is.

Of course it could be argued that these vendor specific features, mainly graphical, but also report writing features, should not be the concern of a JSP debugger. The debugger would be used to debug errors at a lower level than user interaction, and so all unrecognised constructs can be ignored. But how does the interpreter know that these unrecognised constructs are vendor specific instructions and not simply badly written COBOL directives.

This problem is perhaps a little extreme but must be considered by anyone hoping to make a COBOL interpreted diagrammer available to the real world.

This problem aside there are real benefits to be gained from using a JSP diagram level debugger. As mentioned in section 14.3.1 a simple design fault led to an incorrect program which would have required the designer to return to the diagrammer, fix the problem, re-generate, re-compile and then begin testing once more. Using the diagrammer debugger a quick single stepping debugging session revealed the problem, which was quickly altered and tested on-line.

Although this was a small test program in a very limited environment, it seems obvious to conclude that for a more fully featured system, debugging a larger and more intricate program, the benefits would also increase in scale.

## References

[BARSTOW 86] David R. Barstow, Howard E. Shrone, and Erik Sandewall, *Interactive Programming Environments*, McGraw-Hill, 1986

[MEYER 90] Bertrand Meyer, *Introduction to the theory of programming languages*, Prentice Hall, 1990

[MORGAN 90] Carroll Morgan, *Programming from specifications*, Prentice Hall, 1990

[WATSON, DES 89] Des Watson, *High Level Languages and their Compilers*, Addison- Wesley, 1989

[BROWN 89] Judith R. Brown, and Steve Cunningham, *Programming the User Interface - Priciples and Examples*, John Wiley & Sons, 1989

[BURGESS 87] Rod S. Burgess, *Structured Program Design Using JSP*, Hutchinson, 1987

[BENETT 90] J.P. Benett , *Introduction to Compiling Techniques : A first course using Ansi 'C', LEX and YACC*, McGraw Hill, 1990

[AMMERAAL 87] Leendert Ammeraal , *Programs and Data Structures in 'C'*, John Wiley & Sons, 1987

[STORER 87] Ralph Storer, *Practical Program Development using JSP* , Blackwell, 1987

[KERNIGHAN 78] Brian W. Kernighan, and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall,1978

[BROOKS 74] Frederick P. Brooks Jr, *The Mythical Man-Month*, Addison Wesley, 1974

[KRONSJO 87] Lydia Kronsjo , *Algorithms: Their Complexity and Efficiency*, John Wiley & Sons, 1987

[HUNTER 85] Robin Hunter , *Compilers: Their Design and Construction Using Pascal*, John Wiley & Sons, 1985

[TENENBAUM 90] Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein, *Data Structures Using C*, Prentice-Hall, 1990

[AHO 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986

[BERGSTRA 89] Editors: J A Bergstra, J Heering, and P Klint, *Algebraic Specification*, Addison-Wesley, 1989

[GRIES 71] David Gries, *Compiler Construction for Digital Computers*, John Wiley & Sons, 1971

[GOUGH 88] K. John Gough, *Syntax Analysis and Software Tools*, Addison-Wesley, 1988

[BORNAT 79] Richard Bornat, *Understanding and Writing Compilers*, Macmillan, 1979

[HORWOOD 90] W. P. Cockshott, *A Compiler Writer's Toolbox*, Ellis Horwood, 1990

[MAK 91] Ronald Mak, *Writing Compilers & Interpreters: An Applied Approach*, John Wiley & Sons, 1991

[POPKIN 89] Gary S. Popkin, *Comprehensive Structured COBOL*, PWS-Kent, 1989

[BROWN 80] P.J. Brown, *Writing interactive Compilers & Interpreters*, John Wiley & Sons, 1980

[JACKSON 75] M.A. Jackson, *Priciples of Program Design*, Academic Press, 1975

[PARKIN 90] Andrew Parkin, Richard Yorke, and Roger Barnes, *Cobol for Students*, Hodder & Stoughton, 1990

[PRESSMAN 87] Roger S. Pressman, *SOFTWARE ENGINEERING, A Practitioner's Approach*, McGraw-Hill, 1987

[HOLUB 90] Allen I. Holub, *Compiler Design in C*, Prentice-Hall, 1990

[BARCLAY 88] K. Barclay, *Software Engineering Using Executable Specifications*, 1988

[SESSIONS 89] Roger Sessions, *Reusability and data structures in C*, Prentice-Hall, 1989

[WATSON, DEREK 89] Derek Watson, *COBOL for Mainframe and Micro*, Chartwell-Bratt, 1989

[PRATT 84] Terrence W. Pratt, *Programming Languages: Design and implementation.* Prentice-Hall, 1984

[MADDIX 89] Frank Maddix, and Gareth Morgan, *Systems Software: an introduction to language processors and operating systems*, Ellis Horwood, 1989

[TANENBAUM 83] Andrew S. Tanenbaum, Hans Van Staveren, E.G Keizer and Johan W.Stevenson, "A Practical toolkit for making portable compilers", *Communications of the ACM* , September 1983, Vol 26, No 9, pg 654

[NORMAN 89] Ronald J Norman and Jay F Nunamaker Jr, "Case Productivity, perceptions of software engineering professionals", *Communications of the ACM*, September 1989, Vol 32, No 9, pg 1102

[SANDEN 89] Bo Sanden, "An Entity-life modelling approach to the design of concurrent software", *Communications of the ACM*, March 1989, Vol 32, No 3, pg 330

[JALICS 87] Paul J Jalics, "COBOL on a pc: a new perspective on a language and its performance", *Communications of the ACM*, February 1987, Vol 30, No 2, pg 143

[FRITZSON 83] Peter Fritzson, "Symbolic Debugging Through Incremental Compilation in an Integrated Environment", *The Journal of Systems and Software*, Vol 3, No 4, December 1983, pg 285

[ROSSELET 87] A. Rosselet, "Language Definition Based Compiler Development", *The Journal of Systems and Software*, Vol 7, No 2, June 87, pg 145

[OMAN 89] Paul W. Oman, Curtis R. Cook, and Murthi Nanja, "Effects of Programming Experience in Debugging Semantic Errors", *The Journal of Systems and Software*, Vol 9, No 3, March 1989, pg 197

[McCRORY 92] Stephen McCrory, "COBOL: The language they couldn't kill", *Program Now* , January 1992, pg 39

# Part IV
# Appendices

## A COBOL Interpreter User Guide

The program should first be designed using Probench.

When creating the program it should either be called 'altest' or rename all the program files i.e. '.cnd', '.dat' etc to 'altest.cnd', and 'altest.dat'. This has to be done because the programs that prepare the data for the interpreter do not take any arguments and expect the program to be called altest.

### A.1 '.dat' File Conventions

All variables and keywords must be in capital letters, and all valid COBOL statements must end in fullstop.

The only variable types are PIC 9, and PIC X. There should be no use of any COMP or VALUE constructs.

Only 01 through 50 levels are supported, ie no 88's or 77's. There are also no REDEFINES.

### A.2 '.ops' File Conventions

Only 3 COBOL operations are available. These are MOVE, ADD and DISPLAY, the restricted syntax of each of these operators is listed below.

#### A.2.1 MOVE

MOVE can operate on numeric values or numeric variables and transfer these to a specified numeric variable, e.g.

```
001 MOVE 1 TO DNUM.
    MOVE LOOP-VALUE TO DNUM.
```

#### A.2.2 ADD

ADD will add either a numeric value, or the contents of a numeric variable to another numeric variable, e.g.

```
002 ADD 1 to DNUM.
    ADD LOOP-VALUE TO DNUM.
```

### A.2.3 DISPLAY

DISPLAY will display a numeric value, a string or the contents of a variable. It only takes one argument and automatically forces a newline, e.g

```
003 DISPLAY "Hello".
004 DISPLAY DNUM.
```

It is important that each statement end in a fullstop, and all other .ops file conventions are used.

## A.3 '.cnd' File Conventions

Each condition in the '.cnd' file must satisfy the following syntax.

```
CONDITION_NUMBER    ITEM COMPARISON ITEM
```

e.g.

```
C01 LOOP-VALUE > 2
```

For numeric ITEMs, either numeric values or variables, the following comparisons can be used >, <, or =.

For string ITEMs, either a non-numeric literal or a character variable the comparison may only be =.

Both string and numeric comparisons may use the NOT keyword before the comparison operand to negate the condition, e.g.

```
C01 LOOP-VALUE NOT > 2
```

## A.4 Preparing for Interpretation

The following programs should be run in order on drive B.

1. ALLINESC.PRG
2. ALSCAN.PRG
3. SYMCOB.PRG
4. CONDSANY.PRG
5. CONDSBLD.PRG
6. OPSANYL.PRG

7. OPSBLD.PRG

The interpreter can now be used from within the diagrammer as explained in section B.

# B Diagrammer User Guide

The diagrammer allows the creation of JSP diagrams, the allocation of predefined operations and conditions (section A) and the interactive debugging of the diagram.

The diagrammer should be run from drive B and the operations, conditions, and variables will be automatically loaded in.

## B.1 Commands

All the commands take the form of two letters followed by the necessary arguments.

**??** displays a list of all available commands.

**au** Add Under takes as its argument a box number and adds a box under this specified box, example : au 1.

**al** Add Left takes a box number as argument and adds a box to the left of this specified box, example : al 2.

**ar** Add Right takes a box number as its argument and adds a box to the right of this specified box, example : ar 2.

**aa** Add Above, adds a box above the specified box, example : aa 2.

**mu** Multiple Under, takes 2 arguments, the first being the box to add a number of boxes under, the second being the number of boxes, example : mu 2 4.

**ml** Multiple Left adds a number of boxes left of the specified box, example : ml 3 4.

**mr** Multiple Right, the specified box has a specified number of boxes added to the right of it, example : mr 3 4.

**ma** Multiple Above, add a specified number of boxes above the specified box, example : ma 2 3.

**pr** Print Box, when supplied with a box number it will display the box details, example : pr 2.

**nm** Name Box, when supplied with a box number followed by a name, that name is given to the box, example : nm 2 BOX2.

**sh** SHow diagram will display the diagram in a textual representation. Iteration and selection are represented by * & o, operations are listed after the box number.

**dd** Delete box, will delete the specified box number, example : dd 2.

**dt** Draw Tree, graphically draws the tree on screen, although this shows no details it does provide a better view of the structure than [sh].

**eb** Edit Box, a specified box can be defined as having a certain type, and if the parent box is an iteration or selection then it can be given a condition. If the box has no children then it can be allocated operations, example : eb 2.

**do** Display Operations, displays a list of the available operations.

**dc** Display Conditions, will display the available conditions.

**ds** Display Symbols, lists the identifiers and contents.

**rr** Reset Run, will re-initialise all the variables ready for a new trace.

**ss** Single Step, executes the next operation, displaying the operation on screen.

**rc** Run to Completion, runs through the program without interruption, to stop it press the space bar.

**xx** Exits the program.

## B.2 Debugging with the Diagrammer

The diagram should be created using the add operations, with the boxes that need to be anything other than sequence, to have operations or conditions allocated being edited via *eb*.

Once the diagram is complete, run to completion can be initialised by typing *rc*. If the program fails to produce the required results then the run can be reset by typing *rr*, then single stepping through the diagram can be used.

For each step taken the diagrammer displays the operation, the operation number and the current box number.

The state of the variables can be seen at any time by typing *ds*.

By single stepping and checking the variables it should become obvious where the problem lies. The diagram can then be re-edited and tested.

Any run to completion can be halted by pressing the space bar which returns control to the command line.

## B.3 Screenshots

The following screen shots illustrate the diagrammer being used to edit a box, display the diagram, carry out a single step debug, and display the help screen.

```
 Command ?? -help:eb 1
box is type 0
new type is :
0. sequence
1. selection
2. iteration
 => 2

 Command ?? -help:eb 2
box is type 0
new type is :
0. sequence
1. selection
2. iteration
 => 0

box has cond number 0
new cond is => 2

add ops ... 0 to stop => 3
=> 3
=> 0

 Command ?? -help:
```

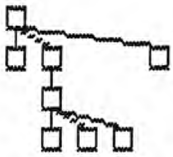
editing boxes using eb

```
 Command ?? -help:eb 0
box is type 0
new type is :
0. sequence
1. selection
2. iteration
 => 0

add ops ... 0 to stop => 5
=> 0

 Command ?? -help:sh

    1
        2 0-2,3,
        3
           *4
               o5 0-4,
               o6 0-1,
               o7 0-2,
        8 0-5,
```



```
 Command ?? -help:dt

 Command ?? -help:
```

Displaying the diagram

```
=> 4
=> 0

 Command ?? -help:ss
BOX  2
OP   1
 MOVE 0 TO DNUM

 Command ?? -help:ss
BOX  2
OP   3
 MOVE 1 TO LOOP-VALUE

 Command ?? -help:ss
BOX  2
OP   4
 ADD 1 TO LOOP-VALUE

 Command ?? -help:ds
1   - LOOP-DATA
3   - DNUM  =  0
3   - LOOP-VALUE  =  2
END of SYMBOLTABLE

 Command ?? -help:
```

A sample debugging run

```
??

  COMMANDS FOR ICD:
au - add under
al - add left
ar - add right
aa - add above
mu - add multiple under , needs total n.o boxes
ml - add multiple left , needs total n.o boxes
mr - add multiple right , needs total n.o boxes
ma - add multiple above , needs total n.o boxes
pr - print box
nm - name box , needs name
sh - show tree
dd - delete box number
db - draw box x , y , size , testing only
dt - draw tree
eb - edit box number
do - display ops,  dc - display conds
ds - display symbols, rr - reset run
ss - single step , rc -run to completion
xx - exit
?? - command list

 Command ?? -help:
```

The help screen

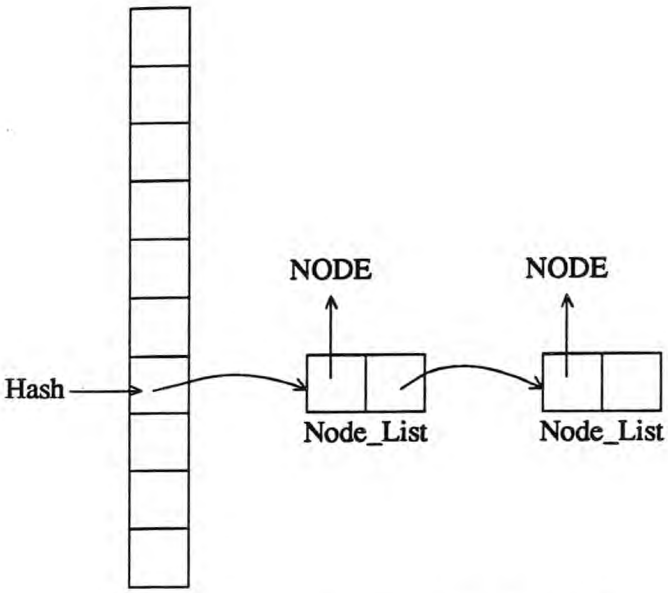



Figure 9: The Symbol Table

# C Data Structures

This appendix details the main data structures used throughout the project.

## C.1 The Symbol Table

The above diagram shows a graphical representation of the symbol table used in the project (see also section G.1.1). The table is based on a dynamically created array. Each array slot contains either a pointer to a NODE_LIST, or a NULL value.

A hash mechanism is used to determine which slot a NODE should be placed in. The actual slot value is determined by hashing the NODE name and using the modulus of the array size; for an array of size fifty the hash value is derived as follows **hash(NAME(node))%50**.

Nodes can be added to a particular slot, and found using either the name or by looking for the node's unique number in a given slot.

The actual code for this structure can be found in section G.1.1 on page 351

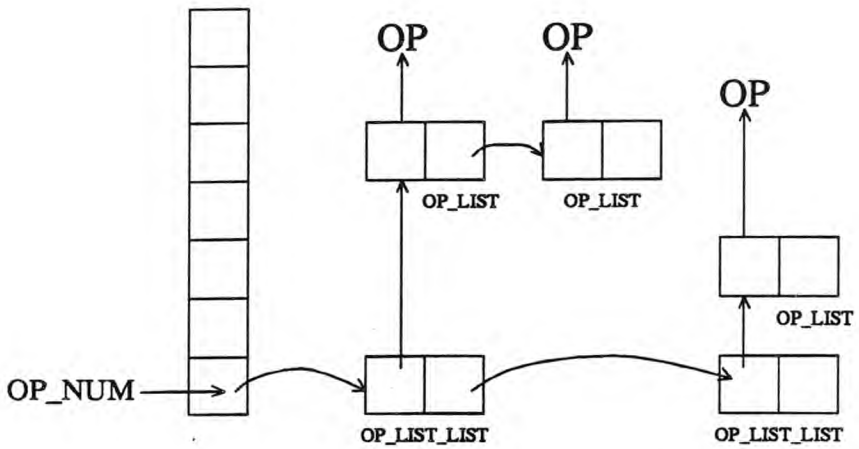



Figure 10: The Operations Table

## C.2 The Operations Table

The operations table (see section G.1.2) is a dynamically created array of OP_LIST_LIST's. An OP_LIST_LIST is a dynamic list of OP_LIST's, this in turn is a dynamically created linked list of OP's.

The above diagram shows this arrangement graphically.

The array is of fixed size, and each slot represents an operation. Each operation can be accessed by its number.

Each OP_LIST_LIST represents a COBOL statement, with the linked list of OP_LIST's being the actual items making up the statement.

An OP can be either a keyword, a variable or a literal.

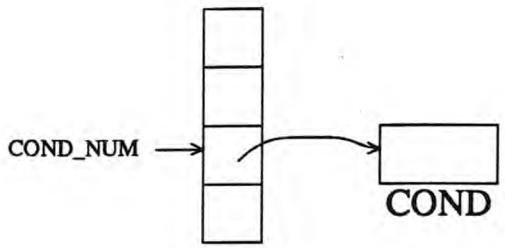



Figure 11: The Conditions Table

## C.3 The Conditions Table

The condition table is based on a dynamic array structure of fixed size. Each array slot contains either a pointer to a COND or a NULL value.

A COND contains a left hand side, a relational operator and a right hand side as shown in section G.1.3 on page 391.

Conditions can be created, evaluated and displayed using the available functions. The entire table can be created, saved and loaded to and from disk.

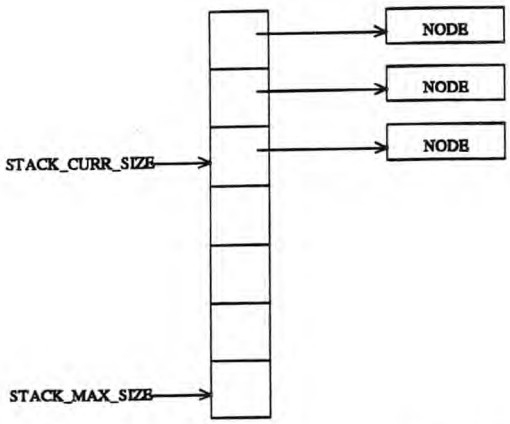



Figure 12: The Stack By Array

## C.4 The Stack

The Stack (section G.1.4) is based on the StackByArray specification presented in [BARCLAY 88].

```
sort            StackByArray[Item]
based on        Bool, Integer, Array
opns
        ABSF : Array * Integer -> Stack
eqns
        NEWSTACK = ABSF(NEWARRAY, 0)
        PUSH(e, ABSF(a,t)) = ABSF(ASSIGN(a, t+1, e), t+1)
        POP(ABSF(a, t)) =
            if t = 0 then ABSF(NEWARRAY, 0) else ABSF(a, t-1)
        TOP(ABSF(a, t)) =
            if t = 0 then ERROR else ACCESS(a, t)
        ISNEWSTACK(ABSF(a,t)) = (t = 0)
```

These functions are reflected by the 'C' functions stackpop, stacktop, stackpush and create_stack_by_array.

The array is dynamically created and contains pointers to NODEs.

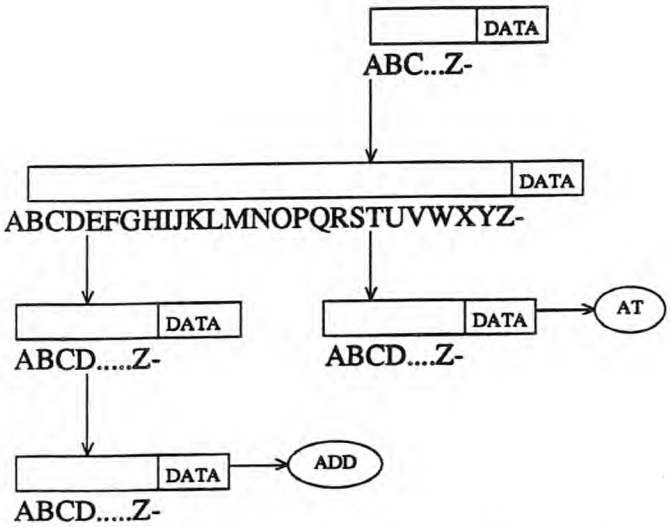



Figure 13: The Trie

## C.5 The Trie

In order to represent the keywords in a manner which could be searched sequentially, using a tree like structure, a prototype structure , tentatively named a staggered list, was built up. Research uncovered that this was a Trie (section G.1.5), and a number of implementations were discovered. The trie used in the project is based on the one presented in [AMMERAAL 87]. This uses a tree-like structure the branches of which are arrays, with 27 pointer nodes and 1 integer value.

Each slot either points to another array or contains a NULL value. In order to search a trie the search string is parsed one character at a time from left to right. This matches perfectly the mechanism where words are read from a file one character at a time.

For each character the array slot for this character is examined, if it contains a pointer then this sequence is valid, if it contains a NULL then the search string is not stored in the trie.

The data value is the number associated with the trie string that ends at this slot.

This structure is heavily memory intensive because even if a link only contains one pointer then 26 other slots have still been granted memory

space.

It would be possible to use linked lists instead of arrays, but this would slow down the searching time, and complicate the data retrieval, but might be the best solution to the memory problem, rather than go back to sequential searching of arrays.

## C.6 The Diagrammer Data Structures

### C.6.1 The Tree

Figure 14: The Tree

The diagrammer tree representation contains a pointer to the root box and an integer representing the total number of boxes.

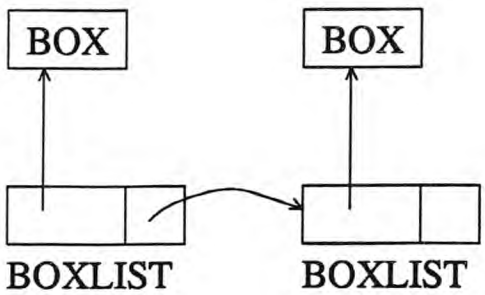



Figure 15: The Boxlist

### C.6.2 The Boxlist

The boxlist is the most formally specified structure used in the diagrammer (section G.2.2). The main reason for this being that the boxlist is the main component used in the construction of the tree. The diagram representation is build using boxlist pointer chains with each boxlist pointing to a box and each box pointing to a linked list of boxlists.

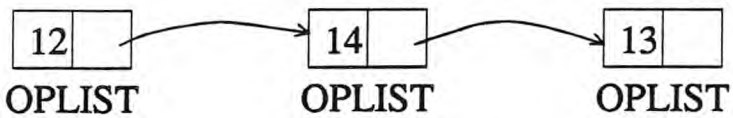



Figure 16: The Oplist

### C.6.3 The Oplist

The Oplist (section G.2.1) is a linked list structure that contains an operation number, and either a pointer to the next operation allocated to a box or a NULL value.

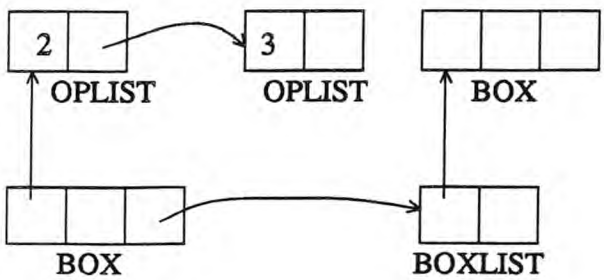



Figure 17: The Box

### C.6.4 The Box

The box (section G.2.2) is the main data repository in the diagram, and it is the structure that contains the starting point for boxlists.

In addition to the box's data the box also stores a pointer to the linked list of operations allocated to the box, and a linked list of boxlist structures representing the box's children.

# D Test Programs

## D.1 Multiply

### D.1.1 Operations

```
001 MOVE 0 TO ANSWER.
002 MOVE 0 TO TNUM.
003 MOVE 0 TO BOTNUM.
004 MOVE 1 TO SUM-COUNT.
005 ADD 1 TO TNUM.
006 ADD 2 TO BOTNUM.
007 ADD TNUM TO ANSWER.
008 ADD 1 TO SUM-COUNT.
009 DISPLAY TNUM.
010 DISPLAY BOTNUM.
011 DISPLAY ANSWER.
012 DISPLAY "---".
013 DISPLAY "*".
```

### D.1.2 Conditions

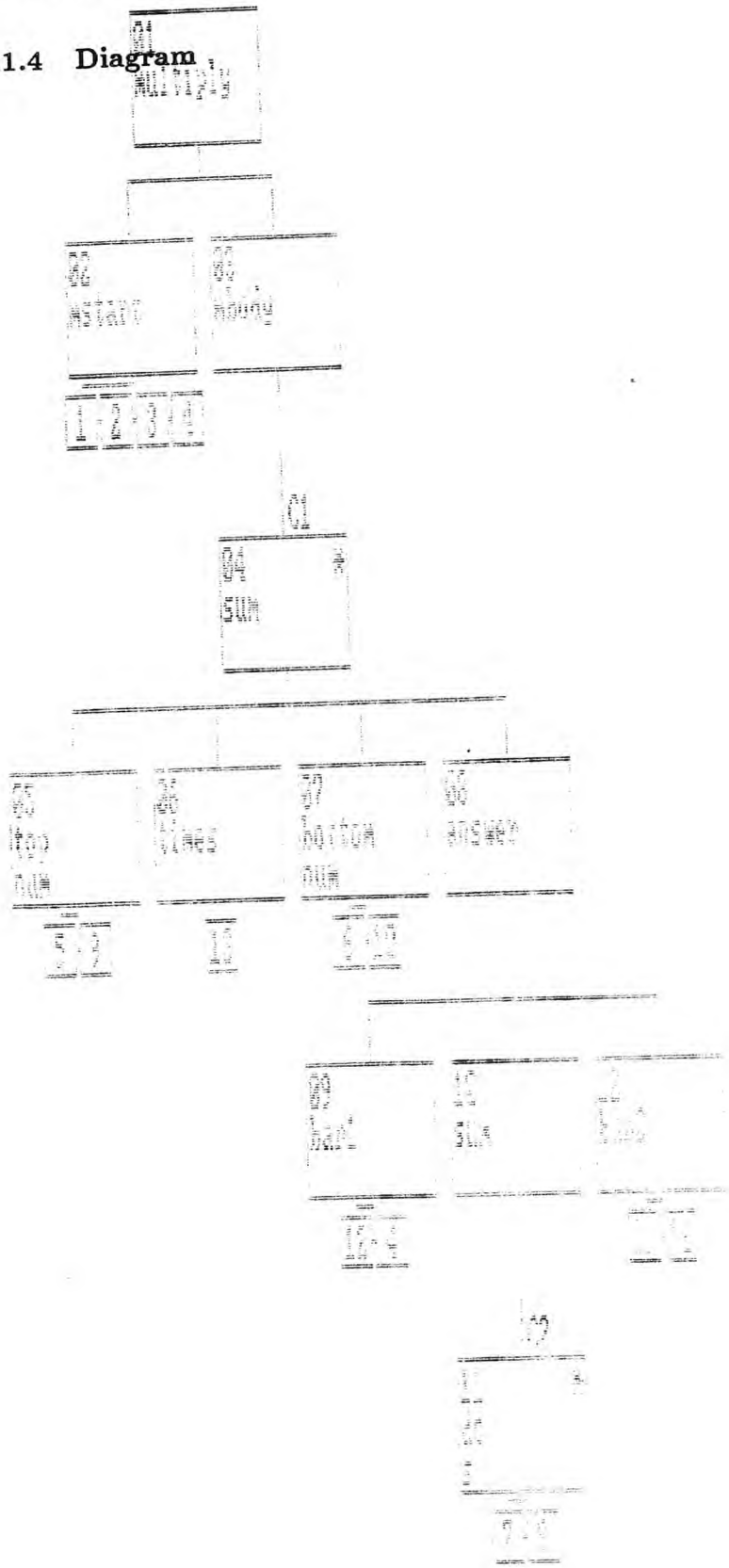
```
C01 TNUM > 10
C02 SUM-COUNT > BOTNUM
```

### D.1.3 Data Definitions

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MULT.
AUTHOR. A. RICHARDSON.
INSTALLATION. NAPIER COLLEGE.
DATE-WRITTEN. 4/3/92.
DATE-COMPILED. 4/3/92.
SECURITY. NONE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MULT-DATA.
   03 ANSWER PIC 99.
   03 SUM-COUNT PIC 99.
```

```
03 TNUM PIC 99.
03 BOTNUM PIC 99.
```

### D.1.4 Diagram


01
MULTIPLY
02
instant
03
hourly
1 2 3 4 5
C1
04
sum
05
top
num
06
times
07
bottom
num
08
answer
5 2
10
1 2
09
half
10
six
11
five
12 4
2 1
C2
1
2
3
4
5 6

## D.2 Text Loop

### D.2.1 Operations

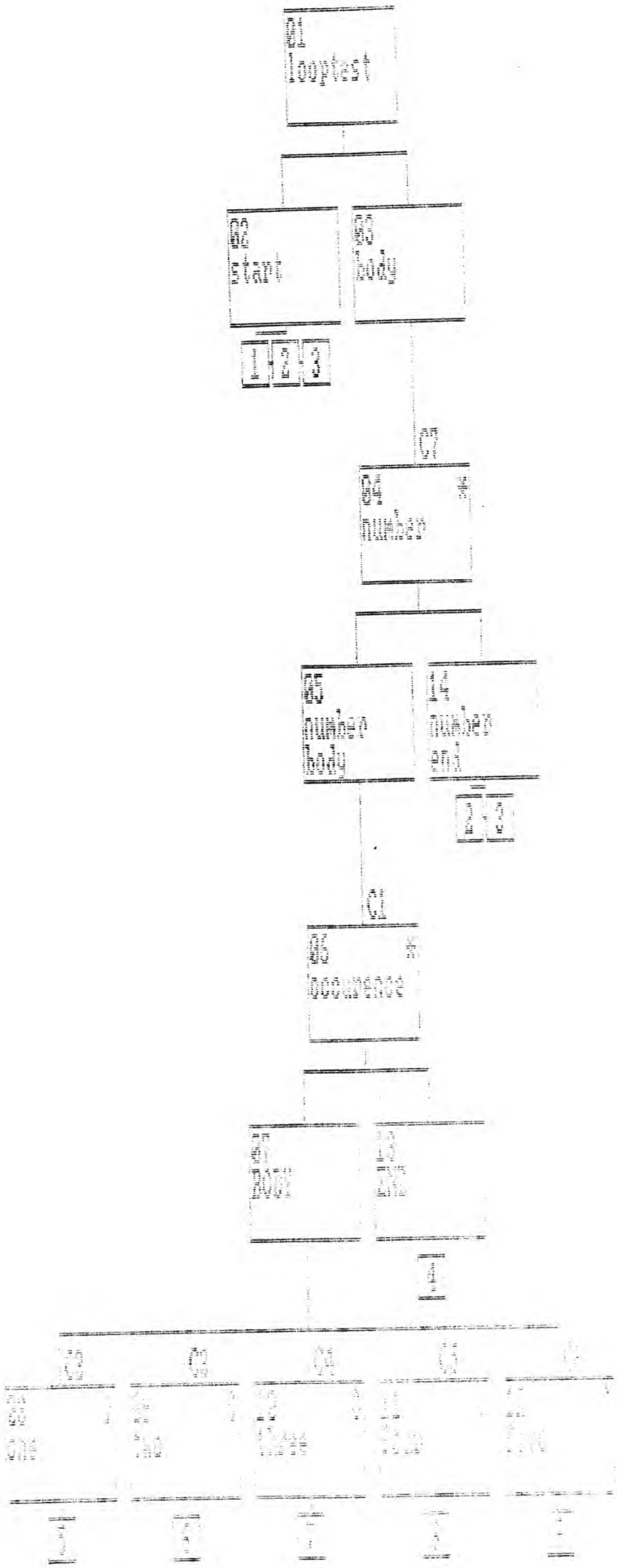
```
001 MOVE 0 TO DNUM.
002 ADD 1 TO DNUM.
003 MOVE 1 TO LOOP-VALUE.
004 ADD 1 TO LOOP-VALUE.
005 DISPLAY "ONE".
006 DISPLAY "TWO".
007 DISPLAY "THREE".
008 DISPLAY "FOUR".
009 DISPLAY "FIVE".
```

### D.2.2 Conditions

```
C01 LOOP-VALUE > DNUM
C02 DNUM = 1
C03 DNUM = 2
C04 DNUM = 3
C05 DNUM = 4
C06 DNUM = 5
C07 DNUM > 5
```

### D.2.3 Data Definitions

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LOOPTEST.
AUTHOR. A. RICHARDSON.
INSTALLATION. NAPIER COLLEGE.
DATE-WRITTEN. 4/3/92.
DATE-COMPILED. 4/3/92.
SECURITY. NONE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 LOOP-DATA.
   03 DNUM PIC 99.
   03 LOOP-VALUE PIC 99.
```


q1
loop/test
q2
start
q3
body
1
2
3
c7
q4
number
*
q5
number
body
14
number
end
2
3
c1
q6
*
begin/end
q7
print
13
end
4
c2
c3
c4
c5
c6
c8
c9
c10
c11
c12
1
2
3
4
5

# E COBOL to C

This appendix describes the probench generated code translator, developed as a necessary side effect of developing the COBOL interpreter.

The development of this product took a substantial amount of project time, as did the non-productive time spent developing without it; and the following text is intended to explain the reasons why it was needed, what it does, and how it works. Also included is the design, code and user manual.

## E.1 Necessity

In the December of 1991 a substantial investment of time had been made into the use of JSP design, coded in 'C', to produce the main processing modules of the interpreter. As mentioned in the design section the intention was to code this in a very sequential and monolithic manner using GOTO's ( the full reasoning given in section 11.1 ). This had proved very successful when inverting the first parts of the symbol table generator together.

The main problem with this approach was that when faced with a larger and more complicated program, namely the symbol table's token analyser, the coding became unwieldy. The resulting code being unmanageable, unmaintainable and bug ridden with very little hope of debugging it.

As mentioned in section 11.1 the program could have been written using a standard 'C' coding approach but this would have caused problems when inverting. This section does mention that the coding should be done as *mechanically* as possible. There already existed a program which could mechanically generate code from a JSP diagram, namely PROBENCH, whose files were being used to drive the interpreter. The only problem being that any operations on a PROBENCH diagram are embedded within COBOL. e.g

```
IF
   *currch==*nextch
 GO TO 1-02-next-box.
```

In order not to waste the design already carried out in the previous weeks, and so that the investigation into the use of JSP for compiler design could be continued; a conversion program had to be designed and written.

## E.2 Design

The programs which would be converted by the translator had to have all operations, quits and conditions written in 'C',

- Operations

```
001 infile=open_input(infile_name);
    eof_marker=fgets(line,100,infile);
002 close_file(infile);
003 poscount++;
004 poscount=0;
005 putc(line[poscount],outfile);
```

- Conditions

```
C01 feof(infile)
C02 feof(infile) || (is_last_full_stop(line) && (line[area2]!='*'))
C03 line[area2]==' '
C04 line[area2]=='*'
C05 line[area2]=='\\'
```

- Quits

```
Q01 (!isupper(currch)) && currch!='-'
Q02 listpointer==NULL
Q03 dat==NULL
```

The ".dat" file normally used for the COBOL WORKING-STORAGE SECTION would be used as the preamble for the 'C' source e.g

```
#include <stdio.h>
#include "alfile.h"
#include "allinesc.h"

#define area2 6
#define infile_name "b:\altest.dat"
```

```
#define outfile_name "b:\\testout.dat"

void main(){

 int poscount=0;
 char line[100];
 FILE *infile,*outfile;
 char *eof_marker;

EOD!
```

The end of the ".dat" is signified by the *EOD!* sequence.

Once this had all been input, PROBENCH was allowed to generate the code and an example is seen in section E.6.6on page 130.

An analysis of this output showed that a number of conversion strategies were needed.

The list can be found in section E.5 on page 98.

Based on this analysis a DSD diagram of the file was created, and a straightforward one pass converter was designed which took one line at a time, converted it and wrote it to the output file.

## E.3 Development

**Pass 1**

Using the design as a set of guidelines a 'C' program to carry this task out was hand coded. This was then tested on the output file and when correct the diagram was expanded to include more features. The process of incremental development was carried out until the translator was at a sufficiently sophisticated level to begin converting itself, ie its own JSP diagram.

As this task progressed certain unforeseen conditions arose, e.g a condition was missed from a selected component which resulted in the following code.

```
IF
   *currch==*nextch
    GO TO 1-02-next-box.
ELSE
    GO TO 1-07-other-box.
```

This was avoided by allocating a condition to the component but would have to be catered for if anyone other than the developer was to use it. The main reason it was being developed in JSP was to allow it to be expanded at a later date.

Once a completed JSP designed converter was compiled, tested and debugged, it was then used in the re-development of the previously completed and inverted programs.

**Pass 2**

As the JSP designed programs grew larger, it became natural to begin to use subtrees. The expectation being that these would be generated as copies into the code. This was not the case. Instead self modifying COBOL code was generated e.g.

```
ALTER 3-001-REST-SUBTREE-A TO
    3-001-EXIT01.
```

In order to convert this it was necessary to split the translation process into two passes. The first pass, essentially a different program, was merged into the existing diagram using correspondence since no inversion was (or is yet) possible.

The new pass scans through the code looking for the subtree code and stores the lables found. The older pass was then modified to do the following.

1. add an integer variable to represent every subtree in the code immediately after the EOD! declarative is found.

```
int g2_001_LINE_BODY_A;
int g3_001_REST_SUBTREE_A;
int g4_001_NORMAL_SUBTREE_A;
```

2. the subtree calls were modified to be an initialisation of the subtree variable to an occurrence number and the subtree was then executed.

```
g3_001_REST_SUBTREE_A=0;
goto C3_001_REST_SUBTREE;
```

3. the subtree exit statement was converted into a conditional goto using case statements e.g

```
switch(g4_001_NORMAL_SUBTREE_A){
case 0 : goto C4_001_EXIT02;break;
case 1 : goto C4_001_EXIT03;break;
}
```

The manner in which the subtree solution has been implemented is adequate, but it could prove to be limiting in certain circumstances. An array of fixed length 50 is used to store the labels, this is adequate for most diagrams but for some this may be a limitation. The exit label could be derived from the label name and the occurrence number and this should be done in future versions.

## E.4 User Guide

There are a number of conventions that should be used when producing 'C' programs on probench for further processing by this utility.

### E.4.1 The Data Definition

The '.dat' file now becomes the preamble of the 'C' code it should be much like the start of your usual 'C' programs with the usual includes and defines. main should be declared as usual with any arguments specified. Everything after the declaration of main should only be declarations of variables, functions can be used provided they are being used to initialise variables. e.g. char *i=(char *)malloc(12);. But there should be no further code.

The reason for this from the translator's point of view is that it may need to insert some extra variables. From the JSP point of view it would be incorrect to do this because you would be carrying out operations that are not reflected by the diagram.

The '.dat' must end with EOD! as the last statement, an example '.dat' is shown below.

```
#include <stdio.h>
#include "alfile.h"
#include "allinesc.h"

#define area2 6
#define infile_name "b:altest.dat"
#define outfile_name "b:\\testout.dat"

void main(){

 int poscount=0;
 char line[100];
 FILE *infile,*outfile;
 char *eof_marker;

EOD!
```

### E.4.2 Conditions

Conditions in the '.cnd' file should follow the normal probench numbering conventions and 'C' constructs should be used.

```
C01 feof(infile)
C02 feof(infile) || (is_last_full_stop(line) && (line[area2]!='*'))
C03 line[area2]==' '
C04 line[area2]=='*'
C05 line[area2]=='\\'
C06 line[area2]=='-'
C07 line[poscount]=='\n'
C08 line[poscount]!=' '
C09 line[poscount]=='\n' || poscount>72
C10 is_blank_or_end(line)
C11 line[area2]=='\''
```

### E.4.3 Operations

Operations are also written using the normal 'C' and probench conventions.

```
001 infile=open_input(infile_name);
    eof_marker=fgets(line,100,infile);
002 close_file(infile);
003 poscount++;
004 poscount=0;
005 putc(line[poscount],outfile);
006 outfile=open_output(outfile_name);
007 close_file(outfile);
009 eof_marker=fgets(line,100,infile);
010 poscount=6;
011 putc('\n',outfile);
```

### E.4.4 Quits

Quits follow the normal probench numbering conventions but the translator requires a different convention, *if* the quit is to have associated operations.

```
Q01 (!isupper(currch)) && currch!='-'
    !Q! printf("error");
       printf("Look no specials");
```

```
Q02 listpointer==NULL
Q03 dat==NULL
```

The very first operation must be preceded by a !Q!, this informs the translator that operations are being used, rather than combining the operations into the condition field.

### E.4.5 Using The Program

The code should be generated in the normal fashion using the Generate Cobol Code option in Probench.

Cobtoc can then be executed as follows

```
cobtoc <infilename> <outfilename>
```

The <infilename> will be the '.cbl' file produced as a result of the code generation. The <outfilename> will be the destination '.c' file.

The program does not generate any error messages and does not produce any error messages, so if the program does not compile then the problem lies with the diagram.

Common problems are selected boxes with no conditions, and subtrees being made from boxes other than sequence boxes.

## E.5 Conversion Strategies

The following conversion strategies are used in pass 2.

### E.5.1 Comments

```
* comment line        =>>    /* comment line*/
```

### E.5.2 Procedure Division

```
PROCEDURE DIVISION.   =>>    PROCEDURE_DIVISION:
```

### E.5.3 STOP RUN

```
STOP RUN.             =>>    goto very_end;
```

very_end is a label placed at the end of the 'C' code just before the final closing brace.

### E.5.4 GO TO lines

```
GO TO <line-label>.   =>>    goto <line_label>;
```

All labels are converted so that '-' becomes '_'.

### E.5.5 Labels

```
<label-name>.         =>>    <label_name>:
```

### E.5.6 ELSE

```
ELSE     GO TO label. =>>    /*ELSE*/  goto label;
```

### E.5.7 IF & IF NOT

```
IF                    =>>     if(

IF NOT                =>>     if(!(

IF NOT NOT            =>>     if(
```

The end of an IF occurs when the goto statement is reached, at which point the necessary number of close brackets are inserted.

### E.5.8 Conditions

```
C01 var == 2            =>>      /*C01*/ var == 2
```

### E.5.9 Quits

```
!Q! printf("error");    =>>      {printf("error");
```

When quits are used, the goto also receives a } around it.

### E.5.10 Normal Lines

```
001 printf("hello");   =>>      /*001*/ printf("hello");
```

## E.6 JSP Design

### E.6.1 Operations

```
001 infile=fopen(*++argv,"r");
002 outfile=fopen(*++argv,"w");
003 exit(1);
004 fgets(inrec,100,infile);
005 fputs(inrec,outfile);
006 fprintf(outfile,"veryend: ;\n}");
007 char_pointer=inrec;
008 putc(*char_pointer,outfile);
009 char_pointer++;
010 putc('/',outfile);
011 /* skip it */;
012 putc('_',outfile);
013 putc(':',outfile);
014 putc('\n',outfile);
015 putc('*',outfile); putc('/',outfile);
016 fprintf(outfile,"goto veryend;\n");
017 fprintf(outfile,"goto C");
    char_pointer+=6;
018 putc(';',outfile);
019 putc('C',outfile);
020 fprintf(outfile,"if(!(\n");
021 loop=0;
022 fprintf(outfile,"))\n");
023 fprintf(outfile,"if((\n");
024 putc('{',outfile);
025 char_pointer+=3;
026 fprintf(outfile,"                    }\n");
027 fclose(infile);
028 loop++;
029 putc('/',outfile); putc('*',outfile);
030 fclose(outfile);
031 infile=fopen(*--argv,"r");
032 check_name[loop]=*char_pointer;
033 check_name[loop]='\0';
034 sub_num=find_sub(check_name);
```

```
035 root[total_subs].sub_name[loop]=check_name[loop];
036 root[total_subs].sub_name[loop]='\0';
037 sub_num=total_subs;
038 total_subs++;
039 sub_num--;
040 check_exit_name[loop]=*char_pointer;
041 check_exit_name[loop]='\0';
042 root[sub_num].exits[root[sub_num].main_number].exit[loop]=
                                        check_exit_name[loop];
043 root[sub_num].exits[root[sub_num].main_number].exit[loop]='\0';
044 root[sub_num].main_number++;
045 loop2=0;
046 loop2++;
047 putc(root[loop].sub_name[loop2],outfile);
048 fprintf(outfile,"                int g");
049 fprintf(outfile,";\n");
050 char_pointer=check_name;
051 putc('=',outfile);
052 exit_num=find_exit(check_exit_name,sub_num-1);
053 fprintf(outfile,"%d;\n",exit_num-1);
054 fprintf(outfile,"                switch(g");
055 char_pointer=root[sub_num-1].sub_name;
056 fprintf(outfile,"){\n");
057 fprintf(outfile,"                }\n");
058 fprintf(outfile,"                case %d : goto C",loop);
059 char_pointer=root[sub_num-1].exits[loop].exit;
060 fprintf(outfile,";break;\n");
061 root[loop].main_number=0;
062 fprintf(outfile,"                g");
```

### E.6.2 Conditions

```
C01 is_it(inrec,"EOD!")
C02 feof(infile)
C03 is_it_blank(inrec)
C04 is_it(inrec,"PROCEDURE ")
C05 inrec[6]=='*'
C06 is_it(inrec,"STOP RUN")
C07 is_it(inrec,"GO TO")
```

```
C08 inrec[7]!=' '
C09 is_it(inrec,"IF NOT") && (!is_it(inrec,"IF NOT NOT"))
C10 is_it_if(inrec)
C11 *char_pointer==' '
C12 *char_pointer=='.'
C13 *char_pointer=='*'
C14 *char_pointer=='\n'
C15 *char_pointer=='-'
C16 is_it(inrec,"GO TO") || is_it(inrec,"!Q!")
C17 is_it(inrec,"!Q!")
C18 loop<5
C19 is_it(inrec,"ALTER ")
C20 sub_num==0
C21 check_name[loop]=='\0'
C22 check_exit_name[loop]=='\0'
C23 loop == total_subs
C24 root[loop].sub_name[loop2]=='\0'
C25 root[loop].sub_name[loop2]=='-'
C26 *char_pointer=='\0'
C27 is_it(inrec,"ELSE ")
C28 loop==root[sub_num-1].main_number
C29 *char_pointer!=' '
```

### E.6.3 Quits

```
Q01 (argc!=3)
    !Q!printf("USAGE- COBOLTOC Inputfilename Outputfilename\n");
Q02 (infile==NULL)
    !Q!printf("cannot open %s as input\n",*argv);
Q03 (outfile==NULL)
    !Q!printf("cannot open %s as output\n",*argv);
```

### E.6.4 Data Definitions

```
#include<stdio.h>
#define NO_SUB_TREES 20
#define SUBTREE_OCCURANCES 30
#define NAME_LENGTH 50
```

```
#define EXIT_LENGTH 30

typedef struct {
                    char exit[EXIT_LENGTH];
                  }EXIT_LIST;

typedef struct {
                    char sub_name[NAME_LENGTH];
                    EXIT_LIST exits[SUBTREE_OCCURANCES];
                    int main_number;
                  }MAIN_ALTER_LIST;
int total_subs=0;

FILE *infile,*outfile;
MAIN_ALTER_LIST root[NO_SUB_TREES];


static int is_it(char *inrec,char *check){

     char *chk;
     chk=check;

     while(*inrec==' ')
          inrec++;

     while(*chk!='\0' && *chk==*inrec){
          chk++;inrec++;
          }

     if(*chk=='\0')
          return(1);
     else
          return(0);
     }

static int is_it_blank(char *inrec){

     while(*inrec==' ')
          inrec++;
```

```
     if(*inrec=='\n')
          return(1);
     else
          return(0);
     }


static int is_it_if(char *inrec){

     char check[]="IF ";
     char *chk;
     chk=check;

     while(*inrec==' ')
          inrec++;

     while(*chk!='\0' && *chk==*inrec){
          chk++;inrec++;
          }

     if(*chk=='\0' || (*inrec=='\n' && *chk==' '))
          return(1);
     else
          return(0);
     }

static int find_exit(char *name,int sub){
     int loop=0,found=0;

     for(loop=0;loop<root[sub].main_number;loop++)
          {
          if(!strcmp(name,root[sub].exits[loop].exit)){
               found=loop+1;
               loop=root[sub].main_number;
               }
          }
     return found;
     }
```

```
static int find_sub(char *name){
     int loop=0,found=0;

     for(loop=0;loop<total_subs;loop++)
          {
          if(!strcmp(name,root[loop].sub_name)){
               found=loop+1;
               loop=total_subs;
               }
          }
     return found;
     }


void main(int argc, char *argv[]){

     int loop,loop2,sub_num,exit_num;
     char inrec[100],*char_pointer;
     char check_name[NAME_LENGTH],check_exit_name[EXIT_LENGTH];


EOD!
```