PROJECT REVIEW:
A COBOL Interpreter in PROBENCH Using 'C'

Student: Alan Richardson
Supervisor: Ralph Storer
BSc Computing Year 5 (Hons)

29th November 1991

Contents

1 INTRODUCTION

This project was chosen because I was interested in compilers, CASE tools, design (both JSP & Formal methods), and I wanted experience with advanced 'C' coding.

This project although it is written to help produce Cobol programs for a DP environment will paradoxically prevent me from having to work in such an environment when the course ends.

There are also a number of improvements which I feel need to be made to the current JSP design environments available, the main one being integration with a Cobol compiler and a higher level of source debugging, considering that the source is the diagram. This project will hopefully highlight areas which can be improved.

2 SPECIFICATION

5th Year Project

Student: Alan Richardson

Supervisor: Ralph Storer

Subject: COBOL interpreter in Probench using 'C'

2.1 Objectives:

The project involves the development of a Cobol interpreter which can be used from within Probench. Since Probench is a working diagrammer already this will have to be done through a series of hooks into the program.

Because of the small time scale the input to the interpreter is assumed to be syntactically correct. This will necessitate the use of a COBOL compiler before the interpreter can be run.

A Cobol program in Probench is built from:

- A .dat file which contains all the data definitions, and file definitions used within the code.
- Conditions used in if statements, and iterations.
- Operations, Cobol instructions which carry out processing tasks.
- Quits, used in backtracking, these define a condition and if it is met then processing is given over to the alternative choice. In Probench these can also have operations associated with them.

Before animating the diagram the interpreter will first of all process the '.dat' file and generate a symbol table. It will then parse the operations, quits, and conditions, cross referencing these with the symbol table.

2.2 Features

The interpreter should have the following facilities :

- Single stepping of program with graphical updates

  This will be done in much the same way as the trace facility with the boxes being inverted as they are encountered, the operations being inverted as they are encountered with the current cobol statement being shown on-screen.

- consulting the state of all the variables at any time during interpretation.
- Abends will be controlled and reported to the user.

3 CHANGES FROM THE INITIAL SPEC

> "Therefore those who are not thoroughly aware of the disadvantages in the use of arms cannot be thoroughly aware of the advantages in the use of arms" *TheArtOfWar,SunTzu*

3.1 First Draft

Initially the project was to be the implementation of a diagrammer which allowed the creation of JSP diagrams, the on-line editing and checking of operations and the on-line tracing of the diagram.

3.2 First Draft - REALITY

Throughout the summer research was conducted into compilers and, since some computer access time was available, tentative work on the diagrammer was carried out.

The diagrammer was pursued to the stage that low level operations had been created to allow the deletion and addition of boxes. However in order to bring the diagrammer up to a stage that could be useable by anyone other than the programmer and to be aesthetically pleasing at the same time, is a project in its own right. Consequently the diagrammer was dropped. Instead the project would be hooked into Probench - an existing, useable and aesthetically pleasing JSP diagrammer.

The time spent on this stage of the project should not be discounted or viewed as wasteful. Since it provided valuable experience into advanced 'C' techniques such as structures, dynamic memory allocation, files, data types. It also provided a means to help the author remember C's idiosynchrasies, and certain stumbling blocks that would have to be faced up to early on in the development stage.

3.3 Second Draft

Having decided to use probench the initial idea was to provide on-line checking of operations, quits and conditions (Probench already provides on-line editing).

3.4 Second Draft - REALITY

Probench was examined and it quickly became apparent that in order to provide these features it would be necessary to rewrite the editing operations to make it more amenable to on-line checking. Also to enforce the creation of a complete, and syntactically correct '.dat' before any such editing was done, thereby constraining the functionality of Probench. In addition new menu items would be

added to allow the batch processing of these files, and all this before the serious business of compiler writing and interactive tracing had begun.

Consequently, this idea was scrapped and instead one new option would be added to probench 'online-trace' (or an equivalent) this would take the .dat, ops, quits, and conditions for a diagram, check them and then begin to interpret the diagram under user control.

3.5 Third Draft

Error checking - The implementation of full error checking of the .dat, operations and conditions with full error recovery - not just panic.

3.6 Third Draft - REALITY

This could be done in the time allocated however no time would be left for the interesting aspect - an animated diagram executing under user control.

Therefore the assumption will be made that all the inputs to the program are syntactically correct, and can be cross referenced perfectly (ie. they could be compiled without syntax errors).

The project design will be done in such a way as to allow this to be added at a later date.

3.7 Final Draft

The project is now at a workable stage, batch processing of .dat, ops, conds and quits. The transformation of these into a form that can be interpreted and animated on-line within probench.

This will involve the creation of parsers for the .dat, ops and conds, each seperate because they embody different syntactical and semantic rules.

A hook into probench to allow the user to choose this and the writing of an on-line trace features in probench.

The functionality of the project is now time dependant is to how much of the interpreter can be built up.

4 APPROACH

"The terrain is to be assessed in terms of distance, difficulty or ease of travel, dimension and safety" $TheArtOfWar, SunTzu$

4.1 Why Cobol

Cobol and JSP are traditionally used together and indeed all the currently available commercial JSP diagrams e.g. Probench & PDF (Program Development Facility developed by Jackson) , use cobol.

Cobol is still used and as such is still taught, often in conjunction with JSP and a Cobol interpreter/diagrammer would be useful during the learning stage.

4.2 Benefits of interpreters

Compilers produce fast code, a good compiler can produce code that can often match that of an intermediate assembly language programmer. So given this, why produce something that executes a program at perhaps $\frac{1}{10}$, $\frac{1}{20}$ or even $\frac{1}{100}$ the speed.

A typical JSP development proceeds as follows design, code, error, then debug. At this point an experienced designer should be able to pinpoint the error quite quickly, they modify the design, recode and check for more errors.

The novice does not have quite the same insights as the more experienced designer and the error messages produced are unintelligible or the results are incorrect for no reason that they can see.

Thus the novice requires some tools to help them out - a debugger. One debugger is unfortunately a native code debugger (assembly or hex) and therefore is of no use. The source level debugger proves to be more helpful and they track down the bug.

This leads to the discovery of another bug, which is fixed and another bug and ... etc. However because the novice has been debugging at source level they have been making changes at the source level and as a result the design has not been changed to reflect this. What they need is a debugger that works at the design level and allows them to track down errors with the diagram therby removing the temptation to edit the source.

It is obvious how this helps the novice, but how does this help the experienced designer? Well it removes the need to look at the program from a different perspective (ie. code), assuming the process can be made to occur fast enough as to not frustrate the user, the interpreter can help the designer track down the elusive errors e.g misplaced operators.

The interpreter is also useful for teaching the novice. The novice can learn JSP constructs :- sequence, selection and iteration, without the need to learn the coding constructs of do - while, while - do & for loops (although these should be learned at a later date) so the onus of a programming course becomes design,

with language operators to allow the design to perform rather than learning a design methodology, and a programming language, then see how they map together.

The main reason for the project is that it allows debugging from within the diagrammer and the user does not need to concern themselves with the code at all.

5 DESIGN

"Therefore victory is not repititious but adapts its form endlessly"
TheArtOfWar,SunTzu

5.1 Methodologies

Most formal compiler texts present little in the way of design to be used when implementing a compiler. The only common method presented is Backus-Naur form, used to describe the grammar of a language. In the case of Cobol the grammar has already been completed. Design seems to be a combination of techniques that have proved to be correct but the actual procedural decisions are not derived in any discernable fashion.

Because of this the design of the interpreter has been treated as a simple design problem using JSP, with an analysis of the input and output to produce the procedural program.

The research into compilers has been useful with respect to the typical data structures used in a compiler/interpreter, ie the symbol table, tokens etc. However in order to make the project complete, these have been redesigned - in some cases this is equivalent to reverse engineering the psuedocode and data structures presented in the research material. This has been done using algebraic specification.

Formal methods has the advantage that each function acting on a data structure is atomic and therefore no clashes occur using these within JSP.

5.2 Problems with JSP

The problems associated with JSP are mainly problems with the resources available to produce JSP diagrams. By this I mean, there is no difficulty with using JSP as a design methodology - it is fast, useful, and provides efficient code. However there is no product available to generate 'C' code. This has led to the situation where diagrams are transformed into code manually. This would not be a problem if the solutions found were kept seperate with interlinking files between them.

In order to remove these files the process of inversion is used. This results in one program as a controller and the other programs are concurrent programs which are activated and halted by the controller.

At first glance this is simply one program invoking the other programs as functions, however each function does not execute to completion. They proceed until they have data to pass back, but returning the data does not mean that the function has completed. The problem here is that when the function is recalled it must start immediately after the return invocation with the data it was using intact.

5.3 Solution

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 immediatly 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, 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 inverson 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 all 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.

5.4 Design Decision

In an effort to keep the design as pure as possible, and this is the aim throughout the project. To allow the design to dictate code so that coding becomes as mechanic as possible - because if the design is only going to be followed part way, then there is really little point in doing the design at all.

Solution 2 has been chosen, although it uses no 'C' constructs, it results from a structured design- so if ever a diagrammer becomes available to generate 'C' code then this can be used to subsequently modify the code easily.

5.5 Communication between programs

Much in the same way that cobol uses a communication division for inter-program communication, the 'C' programs will communicate by passing a pointer to a communication structure which holds the data of the sub-program.

This is consistant with the view that a sub-program is concurrent with the main program since it is an instance of an action required by the main program and so the program exists, but the state of the program depends upon the main program.

Effectively in an advanced system it would be possible to have one copy of each program and they are each called into being in different states by different controller programs, so that they end up being shared resources.

5.6 Differences from traditional approach

The above decisions have led to an approach which is subtely different from that used by most compiler texts.

Normally the lexical analyser produces output as a result of a call to get_a_token from the syntax analyser. The resulting token is a global variable, as is the source line text. This leads to a very open and loosely knit group of functions.

The JSP approach means that only the line_scanner knows about the source lines as they appear on the file (i.e columns 8 through 72). The tokeniser only knows about source statements, and the syntax analyser only knows about a

sequence of tokens. So each stage sees data in a more abstract form and does not know about the other representations. This results in a very tightly knit group.

This also leads to a number of interesting problems concerning error handling. If the syntax analyser discovers an error how does it print the source line with an indicator showing the position of the error. Obviously it doesn't, so it has to pass back the fact that there is an error and allow the only part of the program that does know about source lines, the line scanner.

5.7 Relationship to previous work

The project is unlike any work previously carried out, although certain aspects of the project are familiar through other means.

JSP and Cobol have been covered since second year, with experience in both gained during the placement year.

The theory of Operating Systems was covered in year two, and this also covered some of the theory behind compilers.

The fifth year of the course is now beginning to touch upon certain features found in compilers e.g. software engineering is dealing with formal methods, and the examples given are symbol tables.

Aside from the course I have always been interested in NLP. Thus parsing has always been familiar, even although very little practical experience was gained.

6 PROGRESS

"Take three months to prepare your machines and three months to complete your seige engineering" *TheArtOfWar,SunTzu*

6.1 Current state

At the time of writing, the project is at the stage of processing the .dat file. Analysis of the .dat has split the processing of it, into three parts:

1. The line_scanner which takes the .dat file and removes any Cobol formatting (i.e only columns 8 through 72 contain operational text).

```
e.g    01 NAME                    PIC X(15)
                    VALUE "ALAN".

   becomes

       01 NAME                    PIC X(15) VALUE "ALAN".
```

2. The tokeniser concerns itself with an analysis of Cobol statements (ie. the output from the line_scanner). It does as much processing as it can as to the recognition of the parts of the statement (ie. non-numeric identifiers, keywords, punctuation and terminators).

3. This information is passed to the token analyser which places the correct tokens into the symbol table. At a later date it will also do the syntax analysis of the code.

The plan of action can be seen on the project schedule (Appendix A).

6.2 Scope for enhancement

It has been mentioned previously that this project is a cut down version of what was initially intended. There is therefore scope for enhancement,

This would initially be to provide a fuller set of the cobol language. Time will not permit the production of an interpreter that executes a large number of statements, the actual number is unknown at the moment.

The design should be expanded to accomodate errors and error recovery, this would be done in the form of a backtrack berween good and bad statements, the error recovery would require more thought.

The diagrammer should be altered to allow incremental interpretation so that each statement when entered or altered on-line is partly interpreted instantly. This would reduce the time taken between selecting an on-line trace and actually doing an on-line trace.

The interpreter could be expanded so as to produce object code, or assembly source that can easily be converted to an executable program in the knowledge that it has been tested without ever leaving the diagrammer.

It is doubtfull that the on-line trace will be able to handle backtracking or inversion so these will also need to be covered.

7 APPENDIX A - Gantt Charts

	Month										
Task	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar
Specification											
Compiler Research											
Diagrammer											
— Design											
— Code											
Interpreter											
— Parse .Dat											
— Parse Ops & Conds											
On-line Trace											
Testing											
Documentation											

Table 1: Global Gantt Chart

	Week Ending							
Task	Nov 8	Nov 15	Nov 22	Nov 29	Dec 6	Dec 13	Dec 20	Dec 27
Parse .dat								
— Design								
— 1. Line_Scanner								
— 2. Tokeniser								
— Invert 1 & 2								
— Token Analyser								
— Testing								
— Prepare Project Review								
Parse Ops & Conds								
— Design								

Table 2: Detailed Gantt Chart November/December

8 BIBLIOGRAPHY

Interactive Programming Environments
David R. Barstow,
Howard E. Shrone,
Erik Sandewall
McGraw-Hill 1986

Introduction to the theory of programming languages
Bertrand Meyer
Prentice Hall 1990

Programming from specifications
Carroll Morgan
Prentice Hall 1990

High Level Languages and their Compilers
Des Watson
Addison-Wesley 1989

Programming the User Interface - Priciples and Examples
Judith R. Brown
Steve Cunningham
John Wiley & Sons 1989

Structured Program Design Using JSP
Rod S. Burgess
Hutchinson 1987

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

Programs and Data Structures in 'C'
Leendert Ammeraal
John Wiley & Sons 1987

Practical Program Development using JSP
Ralph Storer
Blackwell 1987

The C Programming Language
Brian W. Kernighan

Dennis M. Ritchie
Prentice-Hall 1978

The Mythical Man-Month
Frederick P. Brooks Jr.
Addison Wesley 1974

Algorithms: Their Complexity and Efficiency
Lydia Kronsjo
John Wiley & Sons 1987

Compilers: Their Design and Construction Using Pascal
Robin Hunter
John Wiley & Sons 1985

Data Structures Using C
Aaron M. Tenenbaum
Yedidyah Langsam
Moshe J. Augenstein
Prentice-Hall 1990

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

Algebraic Specification
Editors: J A Bergstra
J Heering
P Klint
Addison-Wesley 1989

Compiler Construction for Digital Computers
David Gries
John Wiley & Sons 1971

Syntax Analysis and Software Tools
K. John Gough
Addison-Wesley 1988

Understanding and Writing Compilers
Richard Bornat
Macmillan 1979

A Compiler Writer's Toolbox
W. P. Cockshott
Ellis Horwood 1990

Writing Compilers & Interpreters: An Applied Approach
Ronald Mak
John Wiley & Sons 1991

Comprehensive Structured COBOL
Gary S. Popkin
PWS-Kent 1989

Writing interactive Compilers & Interpreters
P.J. Brown
John Wiley & Sons 1980

Priciples of Program Design
M.A. Jackson
Academic Press 1975

Cobol for Students
Andrew Parkin
Richard Yorke
Roger Barnes
Hodder & Stoughton 1990