Durham University

Durham E-Theses

Software maintenance: generating front ends for cross referencer tools

Richard John Turver

How to cite:

Turver, Richard John (1989) Software maintenance: generating front ends for cross referencer tools. Masters thesis, Durham University.

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a https://etheses.durham.ac.uk/id/eprint/6483/ is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
https://etheses.durham.ac.uk

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

Software Maintenance:

Generating Front Ends for Cross Referencer Tools

Richard John Turver

Thesis submitted for the degree of
Master of Science

University of Durham
School of Engineering and Applied Science
Computer Science

11th October 1989

11 MAY 1990

Abstract

This thesis surveys the activities performed in software maintenance and identifies some of the software tools which can be utilised by the maintenance programmer. The most expensive phase of software maintenance is surveyed in more detail and tools to support this activity are identified. A new class of cross referencer tool was designed and investigated. The novel aspect of the cross referencer is that it can be used on more than one language, by utilizing grammar driven generators to customize and make maximum re-use of the language independent components, allowing language specific implementations to be generated with minimal effort. The cross referencer also extends an idea of having different levels of detail in cross reference listings by allowing the tool implementor to specify the contents of each level of detail. A proposed experimental toolkit for the automatic construction of these cross referencer front end tools, from non procedural specifications, is designed and investigated.

"The copyright of this thesis rests with the author. No quotation from it should be published without his written consent and information derived from it should be acknowledged."

Acknowledgements

I am grateful to my supervisor Malcolm Munro for his encouragement and guidance throughout this study. I am grateful to Professor K.H. Bennett for the facilities provided. I am grateful to my parents who have supported me throughout this venture.

Contents

1 Software Maintenance **1**

1.1 Introduction 1

1.2 Software Maintenance Activities 1

1.3 Program Comprehension 3

1.4 Models of Program Comprehension 5

1.5 Tools for Program Comprehension 6

1.5.1 Code Analysis Tools 8

1.5.2 Program Documentation Tools 8

1.5.3 Cross Reference Tools 8

1.6 Summary 9

2 Cross Reference Tools **10**

2.1 Introduction 10

2.2 Commercially Available Tools 10

2.3 Language Independent Tools 13

2.4 Interfaces and Structures 18

2.5 Cross Referencer Internal Structure 18

2.6 Symbol Processing 19

2.7 Summary 21

3 Cross Referencing Language Features **22**

3.1 Introduction 22

3.2 Cross Referencing Different Languages 22

3.3 Programming Language Features 23

3.4 Language Features to be included 35

3.5 Summary 36

4 Application Generators **37**

4.1 Introduction 37

4.2 Application Generators 37

4.3 Compiler and Language Processor Generators 40

4.4 LEX 42

4.5 YACC 42

4.6 Producing Complete front ends 46

4.7 Summary 50

5 A Front End Generator For Cross Reference Tools **51**

5.1 Introduction 51

5.2 Requirements of the Tool 51

5.3 The Cross Referencer Model 57

5.4 An Example 87

5.5 Summary 94

6 Conclusion **95**

6.1 Project Description 95

6.2 Review of Major Points in each chapter 96

6.3 Achievement of Objectives Set 97

6.4 Future Developments 98

A Pascal Lexical Analyser Specification **101**

B A Pascal Parser Specification **105**

C Pascal Cross Referencer Specification **120**

D Lexical Analyser Specification for the Cross Referencer Notation **128**

E Grammar of the Cross Reference Specification **132**

F A Symbol Table Template **139**

G A Generated Symbol Table 141

Chapter 1

Software Maintenance

1.1 Introduction

This chapter surveys the activities performed in software maintenance and identifies some of the tools which can be utilized by the maintenance programmer. The most expensive phase of software maintenance, program understanding, is examined and models of program comprehension are evaluated. Tools to support these models are identified. The importance and use of maintenance tools to capture static information from source code are also discussed.

1.2 Software Maintenance Activities

Maintenance is the most expensive phase of the software life cycle. A survey by Lientz and Swanson [24] showed that frequently more than 50% of the budget was spent on the maintenance phase. It is estimated that there are 77 billion lines of COBOL worldwide and that 30 billion dollars are being spent annually in the USA on maintenance. The US Federal Government owns about 25 billion lines of code and is spending 3.75 billion dollars of its Information Technology budget on maintenance. Software maintenance has traditionally been seen as the tail end product of the software life cycle where the development steps of Requirements, Design, Implementation and Testing have been given

greater prominence. The main reason that there is a maintenance problem is because people regard maintenance as a "necessary evil" and think that software maintenance is just the correction of errors resident in a program when it is released. Planning is also cited as another major problem, people do not think they need to plan maintenance. The name of the discipline is not professionally appealing.

Software maintenance has been defined by the IEEE[2] as:

> *The modification of a software product after delivery to correct faults, to improve performance or other attributes or to adapt the product to changed environment.*

Whilst this definition gives a view of maintenance as any activity performed after delivery, it does not reflect the impact on users by changing the requirements of a software system. Another definition given by Foster[17] is :

> *Software maintenance is the set of activities associated with keeping operational software in tune with the requirements of its users and operators, and all of the people and systems with which the operational system interacts.*

The maintenance activities can be classified by their type[24], as follows:

- Perfective Maintenance which changes the requirements, design and source code of a software system.
- Adaptive Maintenance which changes the design and the source code of a software system.
- Corrective Maintenance which changes the source code of a software system.
- Preventive Maintenance which can change both the design and the code of a system.

Corrective maintenance is defined as bug fixing or the correction of software errors and represents 17% of the software maintenance effort. These are usually emergency program fixes. Adaptive maintenance can be defined as changing the software to deal with environmental changes and accounts for 18% of the software maintenance effort. These usually are changes to data inputs and files, and to hardware and system software. Perfective maintenance is defined as improving

the software's function by responding to customer defined changes and constitutes 60% of the software maintenance activity. Preventive maintenance is defined as enhancing processing system maintainability and accounts for 5% of the maintenance effort. This is done by improvement of documentation and recoding of the source code to produce well structured programs.

Maintenance is seen as the last phase in the conventional life cycle, which consists of Requirements, Design, Implementation, Testing and Maintenance. Given that maintenance is an important part of the software product life cycle we can identify several phases in its own life cycle, thus developing a simple model of the maintenance process. Elements in the model are

- Defining the requirements for change
- Designing the change
- Implementing the change
- Testing the modified software
- Quality assurance on the modification
- Managing the software configuration and classes of the new versions

This simple model reflects a change driven view of the maintenance process where the request for a change invokes the rest of the activities.

1.3 Program Comprehension

Software engineering research resources have been devoted to earlier stages in the software life cycle. This has led to research into formal specification techniques, design methodologies and program correctness proofs. The emphasis in all these approaches has been to reduce the amount of resources dedicated to corrective maintenance. However corrective maintenance only accounts for 17% of the maintenance cost, and perfective maintenance has the greatest cost. Given the simple maintenance model described above, it has been shown that up to 40% of the maintenance effort is spent in trying to understand how the existing software works. This is the most expensive phase of the

software life cycle[39]. It seems worthwhile then to investigate tools and techniques to reduce these costs.

A program as understood by a programmer, consists of a considerably larger body of knowledge than that which is contained in the source code. This information is described in the cognitive psychological terms of knowledge domains. A knowledge domain consists of a closed set of primitive objects, relations among objects, and operators which manipulate these properties or relations. When a programmer completely understands a program, what he knows can be described as a succession of knowledge domains that bridge the gap between the problem being solved and the program in execution. In performing this reconstruction, the programmer makes use of a variety of different sources of information which can be categorised as those which are present in the program text and those which are external to it[25].

The lists below contain examples of the two categories of knowledge. The sources of this knowledge can be from the following:

Internal to the program text:

1. Prologue comments, including data and variable dictionaries.
2. Variable, structure, procedure and label names.
3. Declarations or data divisions.
4. Comments.
5. Indentation or pretty printing.
6. Subroutine and module structure.
7. I/O formats, headers, and device or channel assignments.
8. Action of statements, including organisation.

External to the program text.

1. User's manuals

2. Program logic manuals

3. Diagrammatic representations of programs.

4. Cross reference listings.

5. Published description of algorithms or techniques.

The process of using these two categories of information to reconstruct the knowledge domains is based on successive refinement of hypotheses made by the maintenance programmer and has been reported by Luckey[25].

It is often the case that the sources of information, listed above, are non-existent. A programmer may design a program using structured design techniques but after the program has been tested and altered, the documentation no longer reflects the program[25]. If the program is fifteen years old the documentation may be out of date; for example if one person makes changes and does not record this in the documentation, the next maintenance programmer may discard the documentation if it does not accurately represent the program. If the development programmer did not have maintenance in mind he may not have documented the program at all.

A programmer may only have a program listing as the source of information to build up their understanding. If the program is poorly annotated and poorly structured a great deal of effort will be spent trying to understand how the program works. Methods and tools to support program comprehension, need to be researched in order to reduce the cost of program comprehension.

1.4 Models of Program Comprehension

Most existing models of program comprehension fall into two categories: those based on a code driven approach and those based on a problem driven approach. In the code driven approach, programmers begin by associating small groups of individual instructions with higher level interpretations. These in turn are grouped together and are associated with still higher level interpretations. The understanding process continues in a bottom up manner until overall interpretation is achieved.

The problem driven approach in contrast is a successive top down process of hypotheses generation, and verification. The maintenance programmer starts by forming a primary hypothesis concerning the highest level functions of the program based on whatever information is available. From this hypothesis a cascade of lower level hypotheses are generated and verified against the code. The process continues in this manner until each segment of the program is bound to a subsidiary process.

Knowledge based systems have been implemented to assist and emulate these two approaches to program comprehension. Both these approaches have been tackled in previous projects, the Proust project uses the top down approach and the Pudsy approach uses the bottom up approach.

The Proust approach [38] starts by acquiring high level program goals from the maintenance programmer and then it attempts to match its stored plans with the goals of the program. Plans may contain subgoals. The process is repeated until all the goals have been instantiated.

The Pudsy approach [25] also employs a knowledge base to emulate the problem driven approach by matching standard programming schemas stored in its knowledge base to progressively larger chunks of the program.

Most models of program comprehension, whether bottom up or top down make many assumptions and hypotheses which have not been validated adequately because it is very difficult to prove how a programmer understands a program. However, it is possible to identify certain types of information that a programmer needs to know in order to gain an understanding of how a program works.

1.5 Tools for Program Comprehension

It is possible to identify the sort of information, related to the source code, that a maintenance programmer needs to know in order to change a program even if methods for systematically gaining understanding have not yet been validated. Several categories of tools to facilitate program comprehension have been identified by Wilde[45].

- **Code analysers:** These are programs that statically analyse a program's control structure and data flow. Statically means that the program is not executed, therefore the run time behaviour is not analysed.

- **Documentation aids:** These are tools that generate program documentation that can illustrate the logic at various levels of abstraction. Examples are JSP, Warnier-Orr diagrams and call graphs.

- **Cross referencers:** These tools trace the use of source code objects through a program. Objects are represented by their name and the line number on which they occur. The type of operation associated with the object on the line referenced is also recorded.

- **Restructurers:** These tools accept badly structured program code and produce a well structured program with equivalent functionality.

- **Reformatters:** These tools are intelligent text editors, which enhance program understanding by manipulating the pagination, spacing and indentation of program source code. Reformatters can arrange the source code so that the maintenance programmer can easily depict complex control flow.

- **Execution monitors and debuggers:** These tools monitor and manipulate progress of a program as it executes. They can also be used to determine the effects of various inputs.

- **Test case coverage tools:** This type of tool will tell the programmer which part of the source code was executed for a given set of test data. It associates code segments with user orientated functions.

- **Source Code Comparative Tools:** These types of tools can identify changes between program versions.

The thesis focuses its attention on code analysers, documentation aids and cross referencers, as these tools provide a cost effective way to aid redocumentation .

1.5.1 Code Analysis Tools

Two types of code analysis tools are commercially available, batch oriented metric generators and interactive logic browsers. The output from a batch oriented metric generator output can be compared to predefined standards in order to assess the complexity of a piece of software.

The tools particularly relevant to program understanding are tools such as Via/Insight and FastBol. These can assist a maintenance programmer or software tester to navigate through a program's logic or data flow by isolating specified classes of source statements, such as input conditions, or particular control structures[45].

1.5.2 Program Documentation Tools

These types of tool are useful to the maintenance programmer as they often attempt to represent static information graphically. However many old programs were developed using ad hoc approaches. Therefore if the programs produced were badly structured, a badly structured diagram will be produced by the tool. As well as diagrams, other types of documentation are produced such as file structure analysis (eg Meta Systems -Reverse Engineering). The data structures in commercial software are much larger than technical software, so perhaps tools which focus on data structure redocumentation will be useful to commercial systems in change[45].

1.5.3 Cross Reference Tools

These tools trace the use of names through a program. Object references are identified by source statement numbers. Many cross referencers give additional information such as the way in which an object was referenced (eg where a name is set, assigned to, used in loop etc). The use of a tool to produce static information about the names in a program will make the program understanding process less time consuming and less prone to error[5]. Some tools such as Via/Insight, facilitate on line access to static facts. As software maintenance methods are established, people will become more concerned with tools and cross referencers will become an important item in the maintenance toolkit[45].

Static facts are very useful for input to redocumentation tools as they can provide a large amount of information about all names used in a piece of software. Any tool that requires information of this type will need a front end such as a cross referencer connected to it. A static program analysis toolkit to facilitate program understanding will contain tools such as cross referencers, data flow analysers and control flow analysers. The cross referencer will provide input for data flow and control analysers.

Even after a program has been restructured a maintenance programmer will require a list of all objects and their properties to get some insight into the inputs and outputs of a particular program. A tool that produces static facts may be utilised for constructing preconditions and postconditions.

A toolkit for building front ends could also have other applications for example the collection of metrics. A cross referencer could be made to capture static facts concerning anything in a piece of source code.

1.6 Summary

Software maintenance has been defined and an important aspect, that of program comprehension highlighted. Software maintenance tools were briefly discussed and cross referencers were identified as being a useful tool to facilitate program understanding.

Chapter 2

Cross Reference Tools

2.1 Introduction

This chapter investigates available cross reference tools and identifies their strengths and weaknesses. The state of the art in cross reference tools and the ideal cross reference tool is described.

The difficulties facing cross referencers are investigated and the cross reference tools developed at the Centre for Software Maintenance (CSM) and British Telecom Research Laboratories (BTRL) are evaluated. The importance of language independent cross referencers is identified and the benefits are described. The external and internal structures necessary for cross reference tools are also described.

2.2 Commercially Available Tools

A survey of literature was carried out and a count of the commercially available cross reference tools was made. This gave a total of 44 cross referencer tools and 12 different languages that could be cross referenced[45]. An analysis of this information revealed that

- 24 of the cross reference tools supported only one language.
- 2 of the cross reference tools supported only two languages.
- 3 of the cross reference tools supported only three languages.
- 1 of the cross reference tools supported four languages.
- 1 of the cross reference tools supported five languages.
- 1 of the cross reference tools supported six languages.

12 of the cross reference tools did not specify the language. This may be because they are not fully on the market yet.

The data was then analysed in terms of which language was cross referenced, with the following results.

- Cobol was cross referenced by 16 of the tools.
- Fortran was cross referenced by 7 of the tools.
- JCL was cross referenced by 4 of the tools.
- Basic was cross referenced by 4 of the tools.
- C was cross referenced by 4 of the tools.
- Forth was cross referenced by 3 of the tools.
- RPG was cross referenced by 2 of the tools.
- Pascal was cross referenced by 2 of the tools.
- APL was cross referenced by 1 of the tools.
- Modula-2 was cross referenced by 1 of the tools.
- DBase was cross referenced by 1 of the tools.
- Assembler was cross referenced by 2 of the tools.

This analysis shows that most of the cross reference tools are neither multi-language or language independent.

Most of the cross reference tools commercially available have been built to cross reference one programming language only. Some companies have released more than one version of the same tool, to cross reference other languages. The cross reference tools surveyed are stand alone tools, and it may therefore be difficult to connect them to other tools such as dataflow or control flow analysers.

Many cross referencers provide static facts which can be used by other back end tools for subsequent analysis. One such tool is Via/Insight manufactured by Viasoft. This is a code analysis tool for the programming language Cobol which required ninety man years to build. Much of this tool is based on the Cobol symbol table. If they required to change this tool to process another programming language, apart from having to change the syntax analyser, many of the language dependent components of the tool would have to be changed to accommodate new language features. If the tool were based on some static representation which was language independent then only the front end of the tool would need changing. If the front end analyser were language independent or if it were relatively easy to add new programming languages features, such as new types of objects and a method of representing their properties and relationships to other objects then the tool would become much more versatile as it could be converted to other programming languages in considerably less than ninety man years. Most of the literature surveyed revealed that most tools were language dependent yet most computer installations use more than one language and some installations still produce their specialist languages.

If a company purchases a particular cross referencer tool, for example Cobol, and a new revision of the Cobol language is released then the cross referencer may not cope with the features. The company may have to wait for the manufacturer of the cross referencer to update the cross referencer tool. Altering a language dependent cross referencer to operate on another language would mean a complete re-write. This would mean that the lexer, parser and symbol table would have to be modified to accommodate the different types of grammar rules, different scopes and visibility rules and also the classes of identifier in the language. This would be a very expensive activity.

Some cross reference tools do claim to be language independent, however they are general purpose file searching tools and do not include information about visibility and scope rules that are

features of programming languages. They make no attempt to distinguish between different uses of the same name which may cause the maintenance programmer to make many mistakes. A general file searching cross referencer, whilst being useful, will not provide the maintenance programmer with the complete information required.

The minimum cross referencing tool is nothing more than a general file searching tool that would indicate where symbols were used i.e., the line numbers. The tools might provide some textual context in which the name appears, although the main problem would be that it would not distinguish between variables of the same name in different contexts (different scopes).

Many organisations use more than one language, for specialist applications many companies may have their own language or customise a particular language. Currently a tool is needed for every language. Unfortunately there may not be tools available for specialist languages.

Researchers at the Centre for Software Maintenance have experimented and reflected on the feasibility of producing a language independent cross referencer that would be more than just a general file searching tool. It would provide the same level of detail of static facts that language dependent tools currently produce. This would make the connection to other tools (possibly reverse engineering tools) much simpler as there would be a consistent internal interface to deal with (i.e., a consistent symbol table structure).

This type of system would need to exhibit two main features. There would have to be a simple connection between the source language translator and the tool, or an intermediate representation of the static information. The intermediate representation must cater for all features of a set of languages or must be capable of easily being extended with the minimum of work. The second feature would be a framework to hold all classes of names discovered in a source program.

If a tool like this could be built it would make all the language dependent tools obsolete.

2.3 Language Independent Tools

In order to study the problems of producing language independent cross referencers the internal structure of existing cross referencers had to be looked at. Since the internal structure of commer-

cially available cross referencers are confidential, the study had to restrict itself to those available at the Centre for Software Maintenance and the British Telecom Research Laboratories.

Over a period of time the Centre for Software Maintenance has developed a number of experimental cross reference tools. The objective behind this development was to ascertain the usefulness of cross reference tools for software maintenance and to identify the language independent parts.

The main CSM cross referencer projects were as follows:

1. PXR: a stand alone tool for the language Pascal
2. CXR: a stand alone tool for the language C
3. An Interactive Tool Cross Referencer
4. An Intermediate Output Merger

The **PXR** tool is a context sensitive cross referencer that will produce a cross reference listing of identifiers used in a Pascal program, along with the lines on which they appear. It can differentiate between declarations of the same identifier in different scopes and correctly identify any identifier being used at any time. It will also record the different ways in which an identifier can be used e.g. set, used, called etc. There are two types of cross reference listing available, an alphabetic listing and a structured listing[6].

The cross referencer takes as input a Pascal program together with a set of flags to specify the user options. The user options are designed to let the maintenance programmer regulate the amount of detail which is output from the cross referencer and also the way in which it is presented. There are three flags which can be set:

1. If the 'l' flag is set this suppresses a line numbered listing, otherwise a line numbered listing of the source is produced by default.
2. Flags can be set to specify the level of detail required in the cross reference listing. These flags are 't', 'i' or 'f' for a terse listing, intermediate listing or full listing respectively.
3. Another flag indicates the type of listing required. This can be set to 'a' for an alphabetic listing, in which all identifiers are printed in alphabetic order, or 's' for a structured listing in

which the identifiers are presented alphabetically in sections according to the class of objects represented.

The purpose of the two types of listing is to enable the maintenance programmer to make quick access to an object's static information. A maintenance programmer may only be making changes to a particular section of a program, and therefore will only require a section of the cross reference listing. The purpose of the structured listing is to enable the maintenance programmer to quickly identify a particular section of the cross reference listing. Each section in a cross reference listing has a heading to identify the types of object in that part of the cross reference listing. These headings correspond to the different classes of identifier found in the Pascal Language. The structured listing also distinguishes between new blocks within a function by indenting the information about identifiers in this new block. Some identifiers may be used in specific parts of a program listing and may be affected by change to the code. However they may declared as global variables, which will mean they will not be declared in the same section of the structured listing as variables local to a procedure. Therefore the maintenance programmer can quickly locate those external variables by consulting the alphabetic cross reference listing.

The PXR cross referencer only accepts compilable programs as input. If the Pascal program input to the cross referencer has syntactical errors resident in it, the PXR tool will identify the first error and then terminate cross referencing the source code.

The **CXR** tool is a context sensitive cross referencer for the language C. This tool is very similar to the PXR tool described above. The CXR tool re-uses many modules from the PXR tool[10]. The general philosophy when producing these two tools was to develop a general cross reference tool containing a precise division between language dependent and language independent areas. The CXR tool takes as input, a C program and flags to specify user options. The user options are the same as those in the PXR program. The CXR program produces as output, the two types of listing that the PXR tool produces i.e., structured and alphabetic with three levels of detail of static information terse, intermediate and full.

The **Interactive** cross reference tool provides the maintenance programmer with quick access to specific static information[28]. One of the main problems with the CXR and PXR cross referencer tools is the large amount of cross reference information generated when cross referencing an industrial scale piece of software. It is quite probable that the maintenance programmer will only

need a small amount of information about a program, perhaps locally to the change that is being made in the program. A maintenance programmer may require detailed information concerning one particular data structure or program block. The interactive cross referencer that has been constructed will operate on programs written in the language Pascal, although the tool has been designed so that it can be easily extended to operate on other languages. The tool consists of several parts:

- A **Syntax Analyser** for the language to be cross referenced.
- A **Symbol Table and Symbol Table Routines**. The structure of the symbol table has been implemented as a tree and uses the names of the blocks in the source code as names which make up path names to identify which scope a variable is resident in. Therefore the maintenance programmer can specify a particular block to which queries will relate.
- The **Interface Routines** which extract information from the symbol table and also hide the internal structure of the symbol table.
- An **Interactive Query Processor** which processes maintenance programmers' queries.

The interactive cross referencer has five types of commands for extracting static information:

1. The **WhatIS** command will display information concerning a particular identifier.
2. The **Path** command sets the current search path to the path name given.
3. The **Set** command allows the maintenance programmer to set certain program flags such as output redirection file names and format control flags.
4. The **Print** command allows the maintenance programmer to view the program source text between the line numbers that they specify.
5. The **Calls** command displays the calls made to a routine or the calls out of the routine.

This interactive tool allows the maintenance programmer to gain static information about identifiers used in a program without being swamped with other information.

The **Intermediate Output Merger** tool takes a machine readable file (produced by the front end tool) representing the cross reference listing and merges this file with other files of the same

type. This project extends the functionality of previous projects by allowing the cross referencing of separately compilable software. The merger accepts as input, two intermediate files (produced by a cross referencer) and outputs a single intermediate file. This file contains the combined output of the two input files. The input and output of the merger tool are of the same type so that N files can be merged in N-1 times.

This proved to be a very efficient method of combining cross reference listings, as only one pass is required for each input file and also means that no symbol table has to be built during this merging process[23].

The main BTRL cross referencer projects were as follows:

1. **XREF which is a stand alone tool for the language C**
2. **Multi-language front ends for the languages C, Coral 66, PLM, Assembler 8080**
3. **Intermediate File Output Merger Tool**

The **XREF** cross referencer tool is a British Telecom Development. The XREF cross referencer was designed to satisfy a specific need: to help in the comprehension of a large body of lines (300 000) of largely undocumented code. There are three main sections to the XREF tool:

- The XREF Front end
- The XREF Back end
- The XREF Formatter

The front end contains the language dependent sections of the system. The front end takes as input a program source file and produces as output an intermediate cross reference representation of the source. The output file consists of all the information needed by the XREF back end tool. The back end tool provided by XREF is called XMERGE. This tool takes as input, several intermediate files produced by the front end. The output file contains all the merged input files. This is one solution to dealing with industrial scale software that is written in separately compiled modules. There are two formatters that produce user readable output, XFORMTEX and XFORM. The XFORMTEX produces a file with text formatting commands resident in it, which can then be

subsequently processed by a text processor. The XFORM tool produces a cross reference listing without the textformatter commands resident in it. The XREF tool has been designed so that it can be extended to form part of an interactive system.

The **Multi-language front ends** provide facilities for cross referencing more than one programming language. The original front end to the XREF tool was written for the programming language Coral 66. However the range of front ends has now been extended to include the languages C, PLM and Assembler(8080), to solve the problem of cross referencing software systems written in different languages[22].

2.4 Interfaces and Structures

The need for an interactive cross referencer to control the level of output from a cross referencer was identified by Munro and Robson [28].

In the computer industry there is much research being carried out in the field of human computer interfaces[29,16,14,20] which will certainly have an effect on software tools of the future. Any cross referencer interface would have to be simple and consistent. It may be that future cross reference tools can be dynamically reconfigurable, allowing the maintenance programmer to change the tool as required, by providing pull down and pop up menus, iconic menus and tear off menus, or by utilizing command language macros. Maintenance programmer performance could be optimised by evaluating a suitable screen layout for cross referencers. Different types of windows may be employed e.g., tiles, overlapped, multiple views provided by Suntools, MS Windows, X windows systems and products from Apple, Apollo and IBM etc. However one observation made by Viasoft is that users prefer old interfaces, with which they are familiar, such as the ISPF interface.

2.5 Cross Referencer Internal Structure

The internal structure of an ideal cross referencer should be structured in such a way that the number of interconnections between the components is minimized. This could be achieved by using a combination of stamp and data coupling and also a mixture of functional and informational cohe-

sion between the elements within the cross referencer components. A stand alone cross referencer would probably include the following parts :

1. Main control Handler
2. User Argument Input Handler
3. Compiler Directive Handler
4. Lexical Analyser
5. Syntax Analyser
6. Symbol Processor
7. Listing Generator

2.6 Symbol Processing

The symbol table is a central place where a cross referencer keeps all the information about the user defined names and constants. Often the design of the symbol table depends on the information required by the cross referencer and on how much information can be gained from symbol declarations in a program. The organisation for storing and retrieving elements from a symbol table reflects the scope rules of the language[1]. A compiler uses a symbol table in the same way as a cross referencer to keep track of scope and binding information about names. The symbol table is searched every time a name is encountered. Changes to the symbol table occur if a name or new information about an existing name is discovered[1].

A symbol table mechanism must allow the cross referencer to add new information concerning names in the source code to the symbol table efficiently. There are three common ways of storing symbols namely in linear lists, in hash tables and in tree structures.

Each entry in a symbol table is for the declaration of a name. The format for entries does not have to be uniform, because the information saved about a particular name depends on the usage of the name. Each entry can be implemented as a record consisting of a sequence of consecutive words in memory. These words can be regarded as slots which can hold information. In order to

keep symbol records uniform, it may be convenient to keep some of the information outside those records with a pointer to them.

A declaration of a symbol in a language is a syntactic construct that associates semantic information with a name. Declarations may be explicit or they may be implicit. For example a variable can be implicitly defined in Fortran as an integer if the first letter is an I, unless there is an IMPLICIT statement which overrides this default. The scope rules of a language determine which declaration of a name applies when the name appears in a program construct. The portion of the program in a declaration applies is called the scope of that declaration. An occurrence of a name in a procedure is said to be local to that procedure if it is in the scope of a declaration within the procedure, otherwise the occurrence is said to be non local. A separate symbol table can be kept for each of the scopes. The organisation of the symbol table entries depends on the scope rules of the language:

- Basic can be handled with a table for the outer program and additional separate tables for each subroutine. Variables declared local to a subroutine are only visible in that subroutine.
- Fortran requires a symbol table for each block.
- The programming language Pascal has a strict declare before use rule. Nested scopes are reflected by using a stack as a symbol table, new names are pushed on top of the stack and the stack is appropriately popped at the end of the scope. Whenever a symbol is discovered, the stack is searched top down, this locates the innermost definition of the name.
- Other members of the Algol family do not necessarily require that names are declared before they are used, as long as a declaration occurs within the scope of the name.
- The programming language C permits nested scopes for variables, but functions cannot be nested. There is a declare before use rule, except that functions with integer result need not be declared. A stack of names and a global table of functions will be sufficient for the C language.

This shows that symbol processing mechanism is not uniform in shape and that many types of symbol access rules exist. The programming language features will be looked at in more detail in the next chapter.

The Durham cross referencers PXR and CXR all use the same scheme for holding symbol table information, a tree approach. The tree matches the recursive hierarchy in the attribute grammar of the language the symbol table is designed for. PXR was constructed first although CXR reuses much of the ideas and code used in PXR[10]. The main features of the symbol table tree are as follows:

1. The main structure is a recursive data structure called symbol node which has three components: symbol, less or equal pointer and greater than pointer.
2. A symbol contains the name, owner, block start, block count, symbol class type, the line on which the symbol was declared, and a variant record depending on the class of identifier.
3. The variant record holds all properties of the class of object in question i.e., all the ways in which an object can be manipulated. If one of these variant records is a record for containing a function, sub program or procedure then one of the fields will be a pointer to a new symbol table for all symbols inside this scope.

The syntax analysers are language dependent and the other components are language independent so the CXR and PXR tools can be extended or slightly changed to include other languages[28]. However an enhancement to CXR and PXR will require explicit programming skills and the tool implementor must understand the internals of the symbol table in order to design and implement the new feature.

2.7 Summary

The state of the art in cross reference tools was described and the detailed design of cross reference tools analysed. The language independent components of these tools were also identified.

Chapter 3

Cross Referencing Language Features

3.1 Introduction

Building a software tool that can operate on many different languages requires the tool implementor to understand the evolution of programming languages in order to appreciate why certain features are present in modern languages. Also, in order to identify a set of language features which should be supported by a language independent cross referencer, a survey of languages should be carried out. This chapter examines the main features relevant to cross reference tools, particularly the symbol processing requirements.

3.2 Cross Referencing Different Languages

Tracing the use of objects within a software system can be difficult as each program may be written in a different language. Each language has its own grammar rules, classes of symbols and scope rules. There are commercially available cross reference tools that claim to be language independent[45]. This is because they do not address the different types of language features, instead they are general file searching tools. Fortunately, although two languages may seem different, they often have more similarities than differences. Individual programming languages are not usually built on separate

principles; in fact their differences are often due to minor variations of the same principle[47].

Studies in the way programming languages are used (and misused) have led to a number of design considerations[37]. The influence of these design considerations can be found in the definition of programming languages. Programming languages in general are the result of the realisation that abstraction is the most essential ingredient in programming. Every programming language more or less reflects the depth to which the understanding of the programming activity had evolved at the time of its conception.

There are many languages designed for specific purposes and that is both their strong point and is also often their limitation (eg Algol, FORTRAN - numerical computation, COBOL - data processing). It is unlikely that all the concepts will be integrated in one properly designed language as it would be very complex. Language designers have designed complex languages and then simplified them again (e.g., CPL and BCPL). So perhaps there will always be a need for tools that operate on more than one language.

3.3 Programming Language Features

For a tool implementor to design a tool for current and future languages the historical evolution of languages must be understood in order to appreciate why different features are present. Also there is a huge inertia in the programming environment to the use of new languages, which means that once a language has been successful, it is difficult to supersede it by a newer language. The large investment in established systems ensures the continued use of old languages. This section of the thesis surveys the main programming languages that have been used between the 1950's and 1980's, in order to get a satisfactory knowledge of language features so that a prototype language independent cross referencer could be designed. The following languages have been analysed: **Assembler, FORTRAN, Algol, COBOL, PL/1, BASIC, SIMULA, Pascal, C, Modula-2, and Ada.** For each programming language discussed, the main features are described and the facilities that a language independent cross referencer needs to provide are identified. The features that a language independent cross referencer must provide are summarized at the end of this chapter.

Assembler

Other approaches to programming language processing [33,34,21] do not seem to have addressed software features found in assembly language programs. Many technical, scientific and real-time projects, such as avionic software, have components which are written in assembly code. These software systems may be very large. For example the European Fighter software system consists of 10,000,000 lines of assembler code. Therefore a language independent cross referencer should be capable of dealing with objects found in assembler programs. To cross reference software which runs a particular processor, the architecture of the processor would have to be understood by the cross referencer in order to deal with the instruction set. Although there are many different types of processors they do have some similarities[46]. For example most processors can be categorised by the following criteria:

- The number and type of registers that may be used by the programmer in writing software.
- How the data is organised in memory, and what data types are supported with hardware instructions.
- How memory is addressed by an instruction.
- Special hardware features such as hardware support for stacks.

It is very important to be able to cross reference assembly language programs as they are often called from high level languages. When cross referencing programs written in assembler, the scope and visibility problems of high level languages do not apply. However assembler has many different addressing modes. Often assemblers have built into them macroprocessors. This can cause a problem in deciding whether to cross reference the original program code or the macro expanded code.

To summarize, when cross referencing programs written in assembler the following features will have to be dealt with:

- The use of address registers
- The use of data registers

- The use of the status register
- The data formats in memory
- The many different modes of addressing
- Recording the use of different instructions

FORTRAN

The first attempts to improve machine code were SHORTCODE for the UNIVAC, Speedcode for the IBM 701. The first big breakthrough was the development of 'THE IBM Mathematical FORmula TRANslating system FORTRAN' in 1958[47]. It was developed to allow programming in a mathematical notation which when compiled produced efficient object code which would run faster than a hand coded version. FORTRAN was designed for scientific computations and consequently string handling facilities are non-existent and the only data structure was the array. FORTRAN was a big step forward from SHORTCODE as it included assignment statements that allowed mathematical expression of some complexity on the right-hand side, simple iterative constructs using the DO statement, and subroutines and functions. Formats for input and output were also included. FORTRAN programs are also portable between different machines. One of the main problems in the 1950's was that programmers would not use any other languages and consequently FORTRAN was used in other applications for which it was not particularly suited, such as business data processing. One of the other main problems was that there was no precise standard for FORTRAN. In 1966 this problem was solved when the American National Standards Institute(ANSI) published a standard for FORTRAN. In 1978 a new standard was published, namely FORTRAN 77. FORTRAN data types are INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL and CHARACTER. The cross referencer must be able to store a type associated with each identifier. A cross referencer must also be able to deal with any data structuring facilities. In FORTRAN there is only one, namely the array, which can be up to seven dimensions. A cross referencer must be able to store this in its symbol table. Fortran statements fall into two classes executable and non executable . The non executable statements are declarative in nature. A cross referencer must record all of the declarations of names. The executable statements which affect the use of names must also be recorded.

FORTRAN programs are divided into functional units called functions and subroutines. FORTRAN subprograms may communicate with other subprograms by using a COMMON area or by using parameters. The cross referencer symbol table must provide an area which represents a COMMON area. Also in a FORTRAN program two variables can be made synonymous, sharing the same location in memory by the use of the equivalence statement. This should also be included in a cross reference listing. To summarize, when cross referencing programs written in FORTRAN the following features will have to be dealt with:

- Data types namely INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL and CHARACTER.
- The use of array structures with one to seven dimensions.
- Recording the use of executable and non executable statements.
- The use of functions and subroutines
- The use of arguments and common areas
- The use of equivalence
- The segments of a FORTRAN program are independent of each other. Variables are only visible in the main subprogram where they were declared, unless their visibility is transferred to another part of the program using arguments or the common block.

Algol

Algol has had a large influence on many successive languages for example Pascal, SIMULA 67, C, Modula-2 and Ada[47,44]. Algol was the original block structured language and variables were only valid in the block they were declared in. Arrays can have variable bounds at compile time. There is more than one version of Algol for example: Algol 60, Algol W and Algol 68. The language Algol W was the immediate successor to Algol 60 and contained some new features such as record structures, a case statement, long real and complex data types and complex arithmetic, bits data type and some basic string manipulation facilities. This language was the predecessor to the language Pascal. Algol 68 was produced in 1968 and aimed to be a universal programming

language, suited for scientific and commercial data processing applications. However it made little impact on practising programmers.

A cross referencer would have to support the classical block structure of Algol. Also the cross referencer should support arrays, records and references. Facilities for representing the different parameter mechanisms such as call by value, result, name and value-result would have to be provided. To summarize, when cross referencing programs written in Algol, the following features will have to be dealt with:

- The use of block structure
- The use of data types
- The use of records, arrays and references
- The use of call by value
- The use of call by name
- The use of call by value-result
- The use of call by result
- The use of an identifier need not be declared before it is used provided it is declared within the same block.

COBOL

COBOL is a data processing language, and is quite different from Algol and FORTRAN which emerged around the same time. A COBOL program is divided into four parts:

- The identification division, which is essentially a program prologue which provides commentary and documentation.
- The environment division, which contains machine dependent program specifications. This specifies the connections between the Cobol program and the external data files.
- The data division, which gives a logical description of the data.

- The procedure division which contains the algorithms necessary to solve the problem.

The three main objects found in COBOL programs are files, records and fields.

COBOL, although a language with over 250 verbs, is a relatively easy language to cross reference with the exception of a few features. One difficult feature to deal with is a text substitution facility, a compile time activated utility, the REPLACE statement[3]. Another feature is the redefine command which instructs the compiler that more than one identifier can use the same piece of storage. This is similar to the FORTRAN equivalence statement. This means that more than one name in the source code will be associated with one object in the symbol table. One feature which makes cross referencing simple is that there is only one scope to deal with.

To summarize, when cross referencing programs written in COBOL the following features will have to be dealt with:

- The use of file structures
- The use of record structures
- The use of inter-program communication
- The use of text substitution
- The use of redefinition of identifiers

PL/1

The final version of PL/1 which appeared was based on a combination of ideas found in FORTRAN, Algol and COBOL and was the first attempt at designing a universal programming language. The FORTRAN elements are the parameter passing mechanisms, separately compilable subprograms, common blocks and formated input and output. The block structure and statement structure are taken from Algol. The COBOL features included are the record declaratives[47,35]. A cross referencer would have to support multi-dimensional arrays and structures. Every element in these data structures must be represented in the symbol processing component in the cross referencer. PL/1 has procedures, information is passed between calling subprograms by using parameters and

corresponding arguments. Variables can be declared EXTERNAL in several procedures and each procedure will share the variable in common. This is very similar to the FORTRAN common statement and therefore a facility for representing common variables must be provided by the cross referencer.

To summarize, when cross referencing programs written in PL/1 the following features will have to be dealt with:

- The use of parameter passing mechanisms
- The use of separately compilable subprograms
- The use of common areas
- The use of block structure
- The use of record declarations
- The use of reserved words as identifiers
- The use of multi-dimensional arrays
- The use of structures

BASIC

Basic contains elements of FORTRAN IV in its definition[47] and is a language designed for computers with a small amount of memory and consequently is a small language. Only recent dialects of Basic have included procedures and functions sharing the same visibility rules as other block structured languages such as Pascal. This would be a quite simple language to cross reference, the major problem being the vast number of dialects[47].

To summarize, when cross referencing programs written in BASIC the following features will have to be dealt with:

- The use of procedures and functions

- The use of block structure
- The use of global and local variables
- The use of multi-dimensional arrays
- The use of record structures
- The use of variables that do not need to be declared before use

SIMULA

SIMULA is based on Algol 60 with one very important addition - the class concept[47,31]. The class concept can be found in modern languages such as Concurrent Pascal and CLU. SIMULA classes are based on procedure declarations and the block structure of Algol 60. It is possible to declare a class and generate objects of that class, name these objects and form a hierarchical structure of class declarations. The cross referencer symbol table would have to be capable of holding those hierarchical structures.

To summarize, when cross referencing programs written in SIMULA, the following features will have to be dealt with:

- Classes of objects
- Hierarchical structure of class declarations
- Block structure similar to ALGOL 60

Pascal

The first official language description for Pascal was published in 1971. The Pascal language has elements of Algol 60 and FORTRAN in its design, although it was influenced more by Algol 60. The FORTRAN influence was to make Pascal produce efficient run time implementations. Pascal has static arrays like FORTRAN, although most of the Algol like languages have dynamic arrays, making them less efficient at run-time but more flexible for the programmer. Structured types need

to be considered when designing a cross referencer and these are a prominent feature of Pascal. These types can be built from the primitive types integer, real, boolean and char. The structured, user defined types include arrays, records, sets and files. In addition pointer types may also be used in conjunction with data types. One feature difficult to deal with is the nested record. Field designators are used to descend a level of nesting in the record structure. Another problem is the fact that fields can be given the same name on different levels of record nesting. Therefore the cross referencer must store each level of nesting as a scope and when a nested record structure is being accessed, the correct field identifier must be located. A mechanism is needed to deal with Pascal **with** statement which allows the programmer to reference fields deep within record nesting without specifying a long list of field identifiers. Another problem to be dealt with is the passing of Pascal procedures as parameters.

To summarize, when cross referencing programs written in Pascal the following features will have to be dealt with:

- The use of arrays, sets and records
- The use of procedures and functions
- The use of block structure
- The use of global and local variables
- The use of nested record structures
- The use of pointer types
- The use of the **with** statement
- The use of textual parameterisation

C

C is a programming language developed at AT&T Bell Laboratories around 1972. The C language is made up of functions, data types and program control structures. Like Pascal, C contains pointers and an equivalent of Pascal records, called structures. The basic C data types are numbers which

may be int, float or double; or single characters which are called char. Int numbers may be short, long or unsigned. Each C variable is a storage type and also a storage class (auto, extern, static and register). The cross referencer must be capable of holding these data types and classes. A UNION is a characteristic of C which permits a variable to have different types of values. The cross referencer must be able to hold the definition of each type in the symbol table. A program may be segmented into functions with local variables declared inside them. Function definitions cannot be nested. The cross referencer must store each function in a different scope and allow parameters to be passed to functions. Parameters are, by default, called by value in C, although there is a specific mechanism which allows an argument to be called by reference, by passing the address of the argument, rather than its name.

To summarize, when cross referencing programs written in C the following features will have to be dealt with:

- The use of structures like Pascal records
- The use of unions: variables with more than one type
- The use of data types and storage classes
- The use of functions
- The use of local variables in functions, a scope for each function
- The use of function parameters: call by value or called by reference

Modula-2

Modula-2 is a descendent of the language Pascal[43]. Modula-2 is in effect Pascal with the module concept. The basic Modula-2 data types are numbers, which may be INTEGER, CARDINAL or REAL; logical values, which are called BOOLEAN; and single characters which are called CHAR. A variable need not be declared before it is used. The three internal data structures are called arrays, records and structures.

Modula-2 programs may be segmented by the use of procedures and modules. Modules allow collections of procedures and data declarations to be grouped and separately compiled, and selec-

tively used by other programs as they are needed. A cross reference listing merger tool would be able to merge the cross reference listings produced from each of the module files.

To summarize, when cross referencing programs written in Modula-2, the following features will have to be dealt with:

- The use of data types: INTEGER, CARDINAL, CHAR, BOOLEAN and REAL
- The use of local and global variables
- The use of modules
- The use of nested blocks

Ada

Ada like Modula-2 is also Pascal based, however it is a much more complex language. It extends Pascal constructs but contains features that have no analogue in Pascal. One of the key features is the Package which is designed for the description of large software components and has affinities with the class concept of SIMULA 67 and the module of Modula-2. The procedures and functions are similar to those in Pascal, but the parameter passing mechanism using in, out and in out correspond to Algol W's value, result and value result. Packages allow the programmer to collect several related procedures, functions and type declarations into a single separately compiled unit. Packages have two components: a package specification, which is visible to the user of the package and package body which contains the implementation of the package. The program which uses procedures in the package must be augmented by the prefix

```
with package_name;
```

and all the functions and procedures defined therein are immediately accessible to the program. The cross referencer must have some mechanism for dealing with this facility.

Ada also has tasks to permit parallel processing and contains an extensive set of features for interrupt and exception handling. These are similar to the PL/1 ON-conditions[47,32].

A major problem would be how to cross reference generic packages. Generic program units define a unit template, along with generic parameters that provide the facility for tailoring that template to particular needs at translation time. The template could be for example a queue of items, of some type. The type may not be instantiated until the program is compiled. The notion of generics in Ada is very like macros in conventional assembly languages. A front end will be needed for the cross referencer to do the instantiation.

A cross referencer must associate an identifier in the source code with an object in the cross referencer. A program written in the language Ada can have more than one identifier in the same scope with the same name. Therefore some context other than visibility must be used to distinguish them.

To summarize, when cross referencing programs written in Ada, the following features will have to be dealt with:

- The use of scalar data types: FLOAT or INTEGER and the various forms of precision
- The use of enumeration types: BOOLEAN, CHARACTER or predefined values
- The use of arrays (constrained and unconstrained)
- The use of records
- The use of nested blocks
- The use of global and local identifiers
- The use of packages
- The use of generics
- The use of overloading of previously defined objects

Summary of Historical Survey

Fortran was the first widely used high level language. The next language to appear was Algol 60, the language which greatly influenced Pascal and Ada. The languages Algol 60 , Pascal and Ada are all

block structured. COBOL the main business data processing language allows hierarchical record structures however they are not recursive. PL/1 is a general purpose language which combines many features of FORTRAN, Algol 60 and COBOL. Basic became popular in the 1960's as it was an interactive language and its interpreter would fit into a small machine. Pascal has become very popular as a teaching language as it has structured and primitive types, user defined types, data abstraction and it is block structured. Some dialects of Pascal have information hiding and modularity such MS-Pascal. Modern languages such as Modula-2 and Ada are based on Pascal. Ada includes many more visibility rules than Pascal and can have features such as overloading[42].

3.4 Language Features to be included

If the common features of languages were included in a cross referencer then perhaps a multi-language or language independent cross referencer could be designed. The following features are the characteristics of programming languages which should be included or covered in some way by a language independent cross referencer, since they are found in most programming languages:

- Primitive types
- Structured types
- Manipulation of structured data types
- User defined types
- Data abstraction
- Program block structure
- Visibility rules of names
- Statement and statement sequencing
- Subprograms
- Subprogram sequencing and communication
- Renaming and redefinition of objects

- Generics
- Macro definition and expansion
- Overloading of objects

Chapter 2 described cross referencers such as CXR and PXR which were designed to facilitate the re-use of as much of the program code as possible in a future language dependent implementation of a cross referencer tool. This survey of language features should extend the domain in which a language independent cross referencer can be used. However, the tool implementor will still have to change the language dependent components of the tool. This task will require some effort as the tool implementor will need to understand the internals of the cross referencer tool.

3.5 Summary

A small survey of programming languages was carried out to identify and understand why certain language features are present today. A set of programming language features common to many programming languages, was identified. These features could be supported by a cross referencer tool and would form a language independent nucleus skeleton of a front end to a cross referencer.

Chapter 4

Application Generators

4.1 Introduction

Application generators have been used in many application areas such as user interfaces, databases, commercial data processing and the production of syntax analysers. This chapter investigates the possibility of using a program generator to reduce the amount of effort in producing front ends for software maintenance tools. Few have attempted to generate software maintenance tool front ends, although other researcher's ideas and opinions are evaluated and the important features of their work described.

4.2 Application Generators

Program generators translate specifications into application programs. The specifications describe the work to be performed by the generated program. Specifications of applications programs can be written in a meta language such as Backus Naur Form, expressed graphically or specification can be performed by selecting from a series of menus[9]. Whether the specifications are text or graphics the application generator will generate the program code which is capable of performing the task described. To change or modify the product the input specification would be changed and

processed by the generator again.

The main benefit of application generators is that they offer increased productivity through customized re-usable software. For example, a generator for generating database tables will have re-usable components within it such as functions for generating tuples and domains, assigning types to the tuples and also a dictionary to store the database structure. These main generic parts will have application specific details added to them supplied by the user in some sort of schematic description. The main benefit in this example is that the user does not need to construct these items for each new database and also does not understand the internal details of the generic parts.

Application generators allow the user of the generator to customize and re-use a software design easily. Since software system can be automatically produced, application generators can increase productivity in the design of a product, and will not need to concern themselves with low level design detail. Specifications are far easier to write than the actual program code itself. This is because the user is specifying what is required rather than describing the process needed to achieve what is required. Generators also facilitate rapid prototyping of products. Another advantage is that they provide many projects with a uniform consistent interface and thus can prevent misinterpretations.

The main disadvantage of generators are that they are difficult to build as they require the builder to have knowledge of the application domain and also knowledge of parsers and translators. It is often difficult to identify generic re-usable pieces of program code in the application domain. Generators are often built for very specific application domains and are only useful within those domains. A generator designed for generating a database form interfaces is probably of little use in real time software production.

Generators have been applied to user interfaces for telecommunications equipment, software tools, administrative tools and design processes. Finite state machine based generators use a high level description of the behaviour of a system and produce table driven software. These types of tools have been applied to switching and security software for telephones. They have also been used successfully in the production of parsers for example YACC (Yet Another Compiler Compiler), a LALR(1) parser generator.

Building application generators consists of several tasks such as recognising application domains, defining domain boundaries, defining the underlying model, defining the variant and invariant parts,

defining the specification of the input and implementing the generator[9].

One of the most difficult tasks to be performed when building generators is recognising where to utilize an application generator. The key is to recognise patterns or repetition of constructs in program data structures and program code. Pattern recognition is the most common method used to identify potential application domains.

Defining the domain boundaries can be quite difficult. The generated program may need to connect to other software systems and therefore must interface with it. The difficult aspect of defining domain boundaries is in trying to establish which features should be included and which excluded. Future modifications to the application domain must be anticipated and the application generator must be built in such a way that it can be enhanced easily. The designer of the generator must specify the range of facilities which it is capable of providing.

A generator is likely to be complete, comprehensible and consistent if based on an underlying model. A model will provide the foundation for describing the domain meaning consistently. Each feature can be explained in terms of the model and the specification should be used in terms of the underlying model. Examples of some models are sets, directed graphs, formal logic systems and finite state machines.

Identifying and designing the generic parts or invariant parts is another major step in the process of building a generator. These are features of the generator that do not change and can be re-used by an application. The program generator's variant parts usually correspond to the parts that the user of the generator specifies.

Another feature used for adding variant information is the facility to add user supplied code written in the implementation language. The main problems are that the user needs to understand the implementation language and also how the invariant features work. The feature of adding user supplied code can be used to add functionality to the generator for a variety of unanticipated needs. A drawback of this feature is that it degrades the readability of the specification and also reduces the reliability of the product generated.

Designing the input specification is a difficult task. The design depends upon the application of the generator. Some generators use menus, text, forms, icons and diagrams. Diagrams are used less frequently as many specifications are not naturally pictorial. However some are, for example

networks. It would be interesting to see if the scopes and visibility rules of programming languages could be represented pictorially. For example perhaps the rules that influence a symbol table searching mechanism could be modelled with a simple list of productions. These productions could be generated from a state transition diagram[1] drawn with a diagramming tool.

Implementing the generator is the last task, performed when producing application generators. This task involves writing a program which translates the input into the desired product. The final generator will include a lexical analyser and syntax analyser, a semantic analyser and a product generator. Most generators merge input text, taken or derived from the specification, with templates of program code.

4.3 Compiler and Language Processor Generators

Language processor generators are systems that produce source code analysers for checking the syntax and semantics of computer programs, from a high level specification of the language. The techniques and tools of automatic language implementation have been researched actively for nearly three decades[7]. In spite of the voluminous research effort[13], very few truly usable compiler writing systems have emerged. Probably the most successful system has been YACC[36]. Many later sophisticated systems have not succeeded in breaking through the barrier between the academic and the commercial world.

In order to produce a language processor the first step is to produce a specification of the language. Two aspects constitute a language specification, namely syntax and semantics. The syntax deals with the mechanical aspects, whether or not a sequence of words(or letters) is a sentence in a language. The semantics of a language determine the meaning and legitimacy of the sentence.

Formal notations exist for both parts of the language definition. The syntax is usually defined through a sequence of models, usually in terms of productions. The syntax definition of programming languages is well understood and numerous grammars and notations exist. Describing the semantics of a language is much harder. If a formalism is employed then one method is to simulate the execution of a sentence on a well defined theoretical machine model. The resulting state of the

machine defines the meaning of the sentence, or indicates that it is meaningless.

The syntax of a programming language is usually represented by a context free grammar which consists of a sequence of rules, each rule having a left hand side and a right hand side. Symbols in the rules are terminal or non-terminal symbols. The terminal symbols (lexemes) are the basic building blocks of the language

A grammar defines a language by describing which sentences may be formed. The start symbol of the grammar is a non-terminal symbol from which all possible sentences can be generated. For each non-terminal symbol a production rule must exist, and any one of the formulations of the rule can be substituted for the non-terminal symbol. A sentence is defined to be a sequence of terminal symbols.

It can be seen that a language is a set of sentences, which in turn is a sequence of terminal symbols. Terminal symbols in a programming language usually come in four varieties:

1. operators that are represented as short sequences of characters,
2. reserved words that are represented as sequences of letters whose meaning cannot vary,
3. punctuation marks in the programming language,
4. user defined terminal symbols.

To assemble these tokens from an unstructured stream of input characters a lexical analyser is used.

Lexical analysis is the classical application of the theory of finite state automata[1]. A transition diagram is generally quite easy to derive from a lexical specification of a programming language. An example of a lexical analyser generator is LEX which takes as input a set of regular expressions which defines the terminal symbols or tokens in a programming language and produces a lexical analyser. For example in the language Pascal, an identifier can be defined as follows:

```
{letter}                    [a-zA-Z]
{letter_or_digit}           [a-zA-Z_0-9]
{letter}{letter_or_digit}*  token(Identifier);
```

The **letter** is defined as a single alphabetic character, in upper of lower case. The **letter_or_digit** is defined as a single alphabetic character, in upper or lower case, an underscore, or digit in the range of 0 to 9. The third definition is defined as being a single alphabetic character, in upper or lower case followed by an infinite number of letters or digits. The **token(Identifier)** is a function that returns the sequence of characters or digits when they are detected.

4.4 LEX

A LEX specification consists of three parts:

1. declarations,
2. translation rules,
3. auxiliary procedures.

The declarations section includes declarations of variables, manifest constants, and regular definitions.
The translation rules of a LEX specification consist of a regular expression and an action. These regular expresssions describe which action the lexical analyser should take when the pattern matches the lexeme. In LEX the actions are written in C. The third section holds whatever auxiliary procedures are needed by the actions. Alternatively these actions can be compiled separately and loaded with the lexical analyser.

4.5 YACC

A parser can operate with or without a lexical analyser. If the parser does not use a lexical analyzer it will have to recognise the operators, reserved words, punctuation marks and user defined symbols itself. Alternatively a parser can obtain a series of tokens from the lexical analyser and verify that the string can be generated by the language. This can be thought of as deriving a parse tree. The methods of parsing commonly used in compilers are top down or bottom up. As indicated

by their names, top down parsers build parse trees from the top root to the bottom leaves, while the bottom up parsers start from the leaves and work up to the root. In both cases input to the parser is scanned the same way left to right, one symbol at a time. The popularity of top down parsers is due to the fact that efficient parsers can be constructed more easily by hand using top down methods. Bottom up parsing can handle a wide class of grammars therefore software tools for developing parsers tend to use bottom up methods[18,36].

The top down parsing method is performed by starting with the root, labelled with the starting non-terminal and repeatedly performing the following two steps:

1. At node n, labelled with non-terminal A, select one of the productions for A and construct children at n for the symbols on the right side of the production.
2. Find the next node at which a subtree is to be constructed.

For some grammars, the above steps can be implemented during a single left to right scan of the string. The current token being scanned in the input is frequently referred to as the look ahead symbol. Initially, the look ahead symbol is the first leftmost, token of the input string.

Bottom up parsing is known as shift reduce parsing. Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root. The process can be thought as reducing a string to the start symbol of the grammar. At each reduction step a particular substring matching the right side of the production is replaced by the symbol on the left of that production, and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.

A YACC specification has three parts:

1. declarations,
2. translation rules,
3. supporting C routines.

The declarations part contains the declarations of all tokens that will be passed from the lexical analyser and used in the YACC second and third sections.

The second section contains the translation rules. Each rule consists of a grammar rule and the associated semantic action. In a YACC production the left side is separated from the right side by a colon and alternative right hand sides are separated by a vertical bar. The end of each rule is specified by a semi colon. In a YACC production, a quoted single character is taken to be a symbol, and unquoted strings of letters and digits not declared as tokens are taken to be non-terminal symbols. The left side of the first rule is the start symbol of the grammar.

The third section of the YACC specification consists of supporting C-routines. A lexical analyser by the name yylex() must be provided. The lexical analyser yylex() returns tokens consisting of token type and attribute value pairs. If a token type value is returned, such as DIGIT, the token type must be declared in the first section of the YACC specification. The attribute value is communicated to the parser by a YACC defined variable yyval.

If a language is defined to be a set of sentences, i.e., of sequences of terminal symbols, there are usually many ways to define a grammar describing the language. While the language need not be finite, the grammar must have a finite number of production rules. An important part of the grammar is the recursive constructs which can be used to describe the repetition of production rules.

A grammar for language recognition must not only be unambiguous, but also deterministic. That is the the rest of the input terminal symbols must enable the parser to uniquely decide which rule to use in order to complete the parse tree, if it exists, or to discover that no tree can be constructed to accommodate the next input symbol. There are a number of different classes of grammer that can be written. Some of these lead to parsers which require a number of look ahead symbols.

The grammars of modern programming languages tend to be restricted in form to allow for efficient parsing. They usually possess the LL(1) or LR(1) property, or variations of these. In the first case a parse tree can be built top down without backtracking; in the second case the parse tree can be built bottom up without backtracking. In each case it is only necessary to know, at most the next input symbol to determine which production rule to apply, i.e., one symbol look ahead is required.

A parser specification is prepared for input to the YACC tool and is checked to make sure that

there are rules for all non-terminals. All non-terminals should be reachable from the start symbol and the grammar should satisfy the property of an LALR(1) grammar.

The YACC parser is an LR type of parser and it employs a bottom up syntax analysis technique. L is for left to right scanning of the input, and R is for constructing a rightmost derivation tree in reverse. The driver program is the same for all LR parsers, only the parsing table changes from one parser to another. There are three types of LR parser, SLR, canonical LR and LALR. The SLR is a simple LR parser and is very easy to implement and is the least powerful of the three. The second method, canonical LR, is the most powerful, in that it can describe a large class of languages. The third method, look ahead LR (LALR) is intermediate in cost and power between the other two. YACC is an LALR parser and can be used on most programming language grammars and, with some effort, can be implemented efficiently.

YACC Internals

YACC builds a set of parser tables while analysing the grammar. The parser is a push down automaton. This is a stack machine which consists of a large stack to hold current states and a transition state matrix to derive a new state for each possible combination of current state and the next input symbol. It also contains a table of user defined actions which are performed at certain stages in the recognition and finally an interpreter to actually permit execution.

The internal workings are as follows: the top of the stack contains the current state, the next terminal symbol is produced by the lexical analyser, the parser reads the top of the stack and using the transition matrix selects an operation. The transition matrix consists of five types of operation:

1. ACCEPT: This operation happens only once when a successful recognition is nearly complete.
2. ERROR: This operation happens for all those next terminal symbols which must not be seen in a particular current state.
3. SHIFT NEW STATE: This operation indicates that the next terminal symbol is acceptable in the current state. The new state is pushed or shifted onto the stack and becomes the current state.

4. REDUCE: A reduce takes place when the parser detects a string which matches the right side of a production. This string is replaced by the symbol on the left of the production.

5. GOTO NEW STATE: The reduce action leaves a non-terminal on top of the stack. Goto is the shift operation for this non-terminal symbol. The new state is left on top of the stack.

4.6 Producing Complete front ends

The automation of compiler production has been very popular, however only recently have serious attempts been made to fully automate the process. The recent efforts are the production quality compiler compiler project (PQCC) at Carnegie-Mellon University [30] and the experimental compiler project at IBM. Most other tools are directed at simplifying particular parts of compiler writing , particularly parsing. Little work has been done on generating symbol processing mechanisms as this has become increasingly complex during the evolution of programming languages. Symbol tables are usually based on a model, scheme or some strategy like the Acorn compiler production system developed by Reiss[33,34]. This model must support a variety of entities with different types of names. The scoping rules must also be included. Those two aspects of the model describe the process of mapping an identifier or entity from a piece of source code to an entry in the symbol table. The model is built from a study of the roles which symbols play in programming languages and compilers. The model is different for every programming language. Although some languages do have similar features making certain aspects of the models similar.

The design of the symbol processing facilities is the most difficult aspect of generating front ends. This is because the semantic routines for the various programming languages are not the same and there are no widely used formalisms for describing them. Consequently the resulting features of generated symbol processing mechansims are often of a somewhat experimental character like the HL P84 developed by Koskimies, Nurmi and Paakki[21].

The following describes two existing approaches to the specific problem of generating symbol processing mechanisms. One approach is a model based on set theory and makes no assumptions about the symbol tables to be generated. This makes the specification quite large. Often when generating symbol tables with similar features, the implementor must specify these similar features for each symbol table in which they occur. This suggests that the variant and invariant aspects of

the system were not analysed thoroughly or more likely that a more flexible system was required. The other approach that is discussed is a simple facility for generating language processors which provides a high level of automation. An analysis of these two approaches helped formulate a general design philosophy for reducing the effort required to re-use and customize a language independent cross referencer skeleton.

Koskimies's, Nurmi's and Paakki's general design philosophy was to provide a simple facility with a high degree of automation. They tried to address the problem of generality and whether they should provide facilities for common languages, existing languages or future languages[21]. They concluded that there was no positive answer to this problem as languages are full of particular details. They say that future designers will always design features which are difficult to automate. Their approach was to try and isolate the essence of current languages, design tools to support these essential features, and leave the possibility of resorting to a general purpose language for describing some more elaborate features.

They regarded the following scope features as essential:

1. Entities are defined or declared in the source text before they are used.
2. Entities are associated with textual units that form a tree hierarchy on the basis of textual enclosing.
3. Basic visibility pattern follows this hierarchy, the entities in the current unit and all the enclosing units are potentially visible, in the case of ambiguous naming the entity nearest the unit is selected.
4. The visibility of entities may be restricted up to enclosing units (e.g., modules).
5. The visibility of a set of entities may be transferred from the declarative unit to other textual regions. (e.g., field descriptors, named parameters).
6. The visibility rules may be different for different classes of entities.

Koskimies, Nurmi and Paakki designed their language processor according to these rules. The basic idea is that certain nodes in the derivation tree will be the representation for the entities used in the source text.

The symbol processing component of their language processor stores the entities (called entity nodes) in groups, so that the visibility of a group of related entities can be restricted.

The entity nodes in the symbol table are arranged in a tree structure. Hence the node oriented approach is a natural abstraction of the usual way that symbol tables are constructed in an attribute grammar. The properties of an entity are collected as attributes of the root node of the subtree corresponding to the definition of the entity, and a descriptor of the entity is stored into a symbol table using these attributes. The entities are defined through class definitions. Each class definition defines a class of entities which share the same visibility rules. The entities described in a class may be of different sorts each having a different set of attributes. A class can be viewed as an abstraction of the symbol table.

The visibility of entity nodes is specified with the help of families. A family is a set of entity nodes associated with a branch of their derivation tree. Each family can hold entities of a particular class only. Families are constructed automatically and each entity node will be a member of the nearest family found above the creation point. Families have the visibility options open and closed for regulating the searching mechanism. The search for an entity with a particular key is caused by a pseudo token appearing in a production rule or by a call to a predefined function in an attribute assignment. In either case the searching mechanism looks at the nearest family first then the next family above that one and so on until the entity is found. A closed family breaks off the search. The system also provides a facility called adoption. This means that a family can be transferred from the place where it was created to another place.

This approach to the automatic production of symbol processing mechanisms is general as it only provides common features found in languages. For example, this particular model will not cope with the Ada overloading mechanism. However this model is simplistic and facilitates easy automation. If a particular model tries to include all the divergent scope features in programming languages it becomes very much more difficult to automate and the specification for the symbol processing mechanism becomes longer. A possible other approach would be to analyse a particular class of languages and develop a model of the scope features and visibility rules. This would be complex, and would not be language independent but multi-language. A simple model for a class of languages would facilitate easier automation but the cost would be the loss of generality. This is really the approach that Koskimies, Nurmi and Paakki have taken[21].

Reiss built a model of symbol processing based on the role that symbols play in compilers and programming languages. He attempted to address a wide class of languages and hence his model is complex. Modern languages provide a sophisticated view of symbols and require complex processing[33,34].

The symbol processing mechanism is part of a complete compiler production system. Reiss has accommodated the various roles that symbols play in a compiler and produced a complicated model based on sets. It supports a symbol table to store a variety of information with a variety of access, definition, and control functions. It supports both a view of scoping and supports a signature mechanism for semantic information. The model is based on a set of particular languages, namely FORTRAN, BASIC, Algol 68, Modula-2, Pascal, EUCLID and CLU. This model addresses more symbol processing features than the Helsinki project and it is therefore much more complicated. The specification of a symbol processing mechanism using this model makes the specification large as the generator makes no assumptions concerning the visibility of objects. For example, as all the languages above have an outer scope then it does not appear necessary to specify this feature each time. BASIC, Algol 68, Modula-2 and Pascal can all have block structured scopes, therefore there is no need to specify this when generating symbol processing mechanisms for these languages.

Reiss produced a symbol table package from a symbol table specification. This makes this component a self contained piece of software which is a good feature to have, since a modular system will be much easier to maintain.

The symbol table has a novel searching mechanism which is based on sets of visible objects. It is similar to the block stack mechanism although it has a much more complicated procedure for resolving identifiers with the same name. If a method of adding visibility rules for any unanticipated scope features could be identified this would allow the symbol table to grow. Reiss developed a tool for a specific class of languages although the symbol table will have to change to include future language features.

In conclusion, two approaches have been discussed. One approach aims at a specific class of languages (multi language) and the other approach is a simple facility that may compile FORTRAN, BASIC, Algol 68, PROLOG language processors. However it may not generate all the facilities needed by a language processor. For example it may not deal with FORTRAN equivalence or Ada overloading but may provide most of the facilities needed.

An Abstract View of a Cross Referencer Generator

An application generator was constructed at the Centre for Software Maintenance. Its purpose was to re-use and customise a language independent cross referencer tool skeleton and also to enable language specific implementations of cross reference tools to be produced by a tool implementor with the minimum of effort.

The system is based on a model of symbol processing in compiler front ends and the production of cross reference listings. The approach taken was to produce cross reference tools for a wide class of languages including Ada and also to produce detailed cross reference listings that would be useful to the maintenance programmer. The **meta tool**, that is the tool for generating the cross referencers, takes as input, a nonprocedural specification of the symbol processing and cross reference listing requirements, and outputs compilable source code tool components The source code is the cross referencer tool components. The difference between this approach and that taken by Reiss [33,34] and Koskimies, Nurmi and Paakki [21], is that this cross referencer model has many assumptions built into it concerning the role that symbols play in programming languages. This feature makes the specification much shorter and simpler thus reducing the time taken to produce a new cross reference tool. The tool components produced are then connected to a parser for the programming language to be cross referenced. The parser can be generated using the compiler compiler YACC. The capability of a meta tool such as a cross referencer generator, was illustrated using the language Pascal as an example.

4.7 Summary

Program generators were surveyed and in particular the generation of symbol processing mechanisms of compilers. Two approaches to generating symbol processing modules were analysed and their differences were discussed. A software tool for generating cross reference tools was also described.

Chapter 5

A Front End Generator For Cross Reference Tools

5.1 Introduction

This chapter describes the requirements of a cross referencer tool. It also describes an underlying model for a toolkit for the automatic construction of cross referencer tools. The tools which were constructed are described and an example of the use of the toolkit is also described.

5.2 Requirements of the Tool

The analysis of some computer programming language features, some existing approaches to front end generators and of language independent cross referencer components helped formulate a series of requirements for designing a front end generator for cross reference tools.

The requirements are as follows:-

1. The generated cross referencer should be parser independent.

2. The generated cross referencer should be operating system independent.

3. The generated cross referencer should provide a helpful interface that can either be used in a normal terminal or a windows environment.

4. The output from the cross referencer must represent the contents of the symbol table and be capable of being merged with other output to deal with a multifile environment.

5. The generated cross referencer should produce an output which could be an interface between it and a CASE architecture or used for subsequent processing by another tool.

6. The generated cross referencer should be capable of being used as a stand alone tool.

7. The cross referencer should be capable of dealing with source text which is larger than the available memory by proving an external symbol store. For example if a programmer down loads a large COBOL program for redocumentation or maintenance on a small microcomputer.

8. The cross referencer generator should be simple in design, utilising a series of subtools. Speed and efficiency are not so important since the generator will seldom be used in comparison to other tools like a text editor. The simplicity of design will facilitate easier perfective maintenance of the tool.

9. The cross referencer should consist of discrete components each with a well defined function. For example lexer, parser, variant functions, invariant functions, symbol table and listing generator.

10. The generated symbol table code must not be visible to the parser. Only a set of semantic routines should be visible.

11. The symbol table should be as abstract as possible. Earlier attempts at front ends made the internals of the symbol table visible to the parser. If this processing was hidden from the parser it would simplify the connection between the parser and the cross referencer.

12. The symbol table should be viewed as a series of scopes, consisting of different types of identifiers. Scope sections should be groups of the same class of identifier. Each identifier in the source listing which is being cross referenced will be represented by an object in the symbol table.

13. The cross referencer tool must be based on a language independent skeleton which can be re-used and customised.

14. The variant features which will be added to the skeleton must be added using a generator to reduce the effort required and also to achieve a form of information hiding.

15. The variant features must be specified using a textual schematic definition of the semantic routines needed by the tool implementor. This must be translated into a symbol processing module which maintains the symbol table. The specification must include the class of tokens which designate names of symbols and reserved words. It is not important to specify editing facilities as the source is assumed to be compilable. The specification must describe the entities in the source language, and the routines to enter object usage into the symbol store.

16. The invariant language independent skeleton will consist of facilities for the features common to many languages. It is inefficient to specify this information every time a cross referencer is generated, so the generator will make some assumptions about the language front ends which it is required to build.

17. The cross referencer must be capable of providing different levels of detail in the listings. This should be achieved by passing flags to the cross referencer. The specification of a particular level of detail could be included in the cross referencer specification.

18. The symbol table skeleton must make some assumptions about entities with the same name in the same scope. The new definition can redefine or hide objects; the new definition can overload old objects; the new definition can refer to the old object as it occurs with a forward or incomplete definition; or conflicting names could be an error.

19. Other facilities could be selected by passing flags to the cross referencer, such as whether the cross referencer should distinguish between upper case and lower case identifiers. If there is some aspect of the skeleton which is not required, then a flag may be set which instructs the tool to ignore or include this feature.

20. The skeleton may contain one or more searching mechanisms. It may be possible to influence the search mechanisms by changing the cross referencer specification or by setting some flags.

21. Objects will be associated with a particular class or scope section.

22. Objects will be identified by a path name and its symbolic name in the source.

23. Object sections will be associated with a textual unit or scope. These might be for example packages, modules, functions, procedures, loop scopes or tasks.

24. Scopes can be nested. The visibility follows the nesting.

25. Scopes can be added to the list of visible scopes by using a module or for example by entering a language construct such as the Pascal WITH statement.

26. The visibility of a set of entities may be transferred to another scope by passing parameters.

The design philosophy is to develop a simple facility with a high degree of automation. The approach adopted was to try to generalise a number of concepts into one and to minimize the number of operations and axioms of the system in order to provide a multi language tool based on a few foundations. The project focused on reducing the effort required by the tool implementor to add extra programming language features. A tool implementor could implement a Pascal cross referencer and then might wish to produce a Modula-2 cross referencer. It should be possible to achieve this by changing the specification for the lexical analyser and parser for Pascal and by also changing the specification for the Pascal symbol table. A new tool could thus be produced without the need for additional coding.

The next section describes the prototype cross referencer generator. If this generalised approach is capable of providing the same level of detail of static information and with less effort, then a cross referencer that works for only one language will become obsolete. It may be that the generated code is slower to execute than hand written code but generated code can be changed quickly by running a modified specification through the generator. It is a trade off between speed of execution and ease of maintenance.

If the cross referencer generator model were based on the common features of programming languages, and for example, a language processor were required to be built for a very specialised language then the general approach may only cope with half of the language semantics.

If a cross referencer model is based on a set of eight languages and a very specialised language is not in this set, then the tool implementor may have to hand code the entire symbol processing mechanism for this specialist language. With the common features approach, the specialised language symbol table will not be complete, but a 50% saving will have been made. It will be quicker to use the general approach. If a tool implementor requires to mass produce front ends then the

general approach will be more profitable. If a company uses nothing but COBOL perhaps the specific approach would be more economical.

The general approach taken was to delay commitment as long as possible since even if the design phase fails to produce a successful approach to producing language independent tools, it will have at least led to a better understanding of the problem.

Major design decisions were delayed until all the constraints were known. This was why an extra survey of languages was included in the project. It was very important to perform a language feature survey of some magnitude in order that decisions were not made when there was insufficient information on the symbol processing aspects of languages. When designing the cross referencer model, care had to be taken not to make it too specific. It was also important not to make early design decisions in order to facilitate iterative design. The original generator idea was to have one tool which would perform the generation of the symbol table and semantic routines. This is convenient for the tool implementor using this type of tool. However it was thought that perhaps if the generator was split into a tool kit in which each subtool customised a particular part of the cross referencer it would make the implementation of the tool simpler.

The approach taken by compiler writers could be adopted, that is, subdividing the program code into components with a well defined function. Sub-dividing the task of writing compilers or cross referencers will also provide a basis for re-using these sub-components. As programming languages have common features it is fairly likely that re-use will take place in symbol processing and cross referencing mechanisms.

Overview

The main feature of this new tool is that it is a high level solution to the problem of producing a language independent cross referencer. Detailed cross reference listings will still be produced despite the tool being language independent. This will be achieved by customising and re-using three skeletons, one for the lexical analyser, one for the parser and one for the symbol table. If a change to the tool is required, then the specification is changed, not the underlying code. This will relieve the programmer from understanding the internals of the tool. This approach to language independent cross referencers allows the tool to be configured for different languages without doing

any programming. The diagram following, fig. 1, shows the three skeletons which are to be customised, the lexical analyser, the parser and the cross referencer skeleton. The two sets of tables and the schema are the specifications of the variant parts of the tool. One set of tables is produced for the lexical analyser and the other set is produced for the parser using existing technology. The schema describes the symbol table, semantic routines and listing generator which is labelled as the cross referencer skeleton. The arrows from the lexical analyser skeleton and the parser skeleton represent the parser invoking the lexical analyser and the lexical analyser returning the current token to the parser. The arrow between the parser and the cross referencer represents the function calls made to the semantic routines. The rectangular box labelled as the Cross Referencer Skeleton represents the language independent cross referencer components. All the rectangular boxes link together to form the cross reference tool.

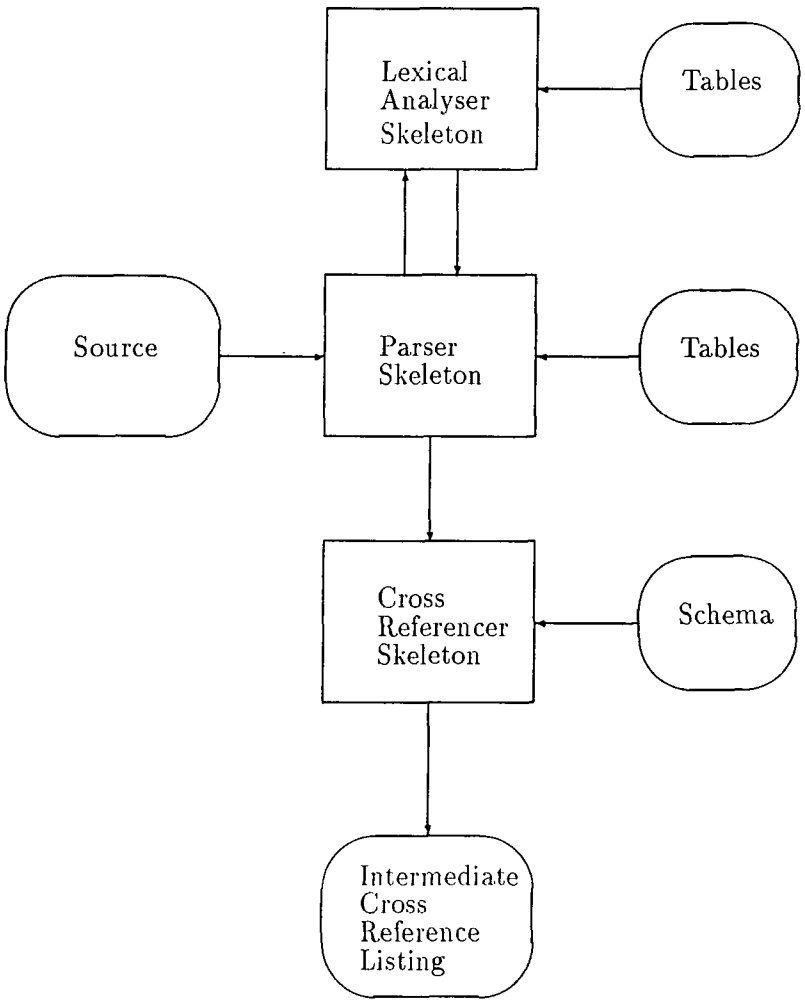

fig.1 A High Level View of the System

The cross referencer tool has this structure as it facilitates easier modification of each of the components because the tool implementor can concentrate on one particular aspect of the system.

5.3 The Cross Referencer Model

Program generators are usually based on some kind of model [9]. The cross referencer model has two main components which are a symbol table and cross reference listing printer. The symbol table can store different classes of symbol, each having different access and definition characteristics as well as scoping features. When activated the printer will extract and output the objects stored in the symbol table together with all of their associated properties. This section of the thesis describes informally the language independent symbol table model. The model is based on language features and symbols in programming languages. It includes common symbol processing methods. The symbol table mechanism may also be able to deal with objects and scopes.

Objects represent identifiers found in source code. The identifier or symbol is the lexical token associated with the object. In programming languages many different types of identifiers can be found and they all have different scope, definition and access rules. When a piece of source code makes reference to an identifier, a cross referencer must determine which object is to be used. This is achieved by using the context in which the identifier has been referenced.

All symbols need to have their usage recorded. One aspect recorded would be a description of the object used. The aspect would be the line number or file name, where the usage was recorded. An object could be stored as a frame of slots. Each slot would contain a meta slot value which would describe the object usage and a slot value which would contain a line number or source file. For example:- for the identifier 'I' defined in a Pascal procedure 'Writename' in the file 'write.p'

META SLOT VALUE	SLOT VALUE
object	I
name space	Writename
class	var
type	integer
declared on	write.p: 11
used on	write.p: 14,15,34
in FOR loop on	write.p: 13

The actual slot value can be single value or multiple slot values and can have an alphabetic or numeric type. Any number of slots can be stored in an object frame. The main structure of an object will be invariant. The variant part which contains the static information will be specified in the tool specification.

This object structure can be viewed as an abstraction of a symbol table and provides a high level view of the storage of symbol information, rather than the parser having knowledge of the detailed physical record structures, as with earlier Durham cross referencers [6,10].

A scope is a region in the source code which defines the visibility and lifetime of an object.

An Index System for the Objects

The model supports two types of index. These indexes are for direct access to the objects in a particular order. The two types of listing are:

- Alphabetic index
- Block structured index

The alphabetic index will be used to produce the alphabetic cross reference listing, while the block structured index will produce a list of objects grouped by their object class. The objects within a particular class are listed in alphabetic order.

Each list contains the object name and its path name. The path name is a list of scope names which are visible from the point where the object was discovered. The listing will contain the pointer to the object in the symbol store and also the class of object, so the printer will call the correct print routine for the different types of objects.

A Mechanism for Remembering Objects

One feature found in many programming languages is the facility to assign the same type to a list of identifiers. For example in the language Pascal:

- VAT,PRICE,POSTAGE: real;

This leads to the problem of not knowing the type of 'VAT' until the end of the declaration is recorded. A solution to this problem can be a symbol stack to contain VAT,PRICE,POSTAGE. When the parser encounters the type identifier, it assigns the text "real" to all objects on the symbol stack. In fact, when an identifier is discovered, an object is associated with it or an object is automatically created and all messages concerning this identifier are sent to this object, until the command disconnect is called. Therefore the symbol stack contains all the objects to which messages are to be sent until the disconnect command is issued.

Scopes can be described in terms of their extent, that is, the region of code an object can be referenced. Visibility rules determine which objects are visible from a particular scope.

Symbol Table Interface

It was observed that some kind of abstract interface was needed between the different cross referencer components in order to reduce the effort to put them together. To make the idea of using a grammar driven program generator to re-use and customise a language independent nucleus attractive, the cross referencer component must conform to some kind of user model of the tool under development. This allows the tool implementor to be relieved of the detailed coding concerns and to view available components in terms of some kind of simplified model of the cross referencer under construction. The internals of the symbol table are shielded from the other components which are part of the cross referencer by using an interface. The symbol table interface is a text file of symbol processing functions which manipulate the symbol table. To make the symbol table interface as simple as possible the source code being cross referenced is assumed compilable(i.e. it contains no syntactic or semantic errors). This reduces the coupling between the symbol table and the parser. The symbol table access routines does not return anything to the parser, as all the decision making is done within the symbol table. As the cross referencer is designed to cope with Ada the symbol table interface will contain some facilities that are not required by other languages. The tool implementor should choose the facilities required and ignore the other functions in the interface. The symbol table interface for the language Pascal would be as follows:

```
Interface CROSS_REFERENCER is;

type CHARACTER_STRING_TYPE is private;
type LINE_NUMBER_TYPE is private;

function INITIALISE_CROSS_REFERENCER();
function CONNECT_TO_OBJECT(Identifier:CHARACTER_STRING_TYPE);
function DISCONNECT_FROM_OBJECT();
function ENTER_SCOPE();
function USE_SCOPE(Identifier:CHARACTER_STRING_TYPE);
function FINISHED_WITH_SCOPE();
function EXIT_CURRENT_SCOPE();
function MAKE_OBJECT_PUBLIC(Identifier:CHARACTER_STRING_TYPE);
function MAKE_OBJECT_PRIVATE(Identifier:CHARACTER_STRING_TYPE);
function ADD_TO_TEXT_BUFFER(Identifier:CHARACTER_STRING_TYPE);
function END_OF_CROSS_REFERENCE();
function PRINT_LISTING();
function PARAMETER_CLASS();
function LABEL_CLASS();
function VAR_CLASS();
function CONST_CLASS();
function TYPE_CLASS();
function FUNCTION_CLASS();
function PROCEDURE_CLASS();
function NAMESPACE(Source_file_name:CHARACTER_STRING_TYPE);
function VAR_AS_PARAMETER(line_number:LINE_NUMBER_TYPE);
function CLASS();
function PASCAL_TYPE();
function FUNCTION_TYPE(Identifier:CHARACTER_STRING_TYPE);
function EXTERNAL_ON(line_number:LINE_NUMBER_TYPE);
function FORWARD_DECLARED_ON(line_number:LINE_NUMBER_TYPE);
function CONSTANT_VALUE();
function NON_LOCAL(line_number:LINE_NUMBER_TYPE);
```

```
function DECLARED_ON(line_number:LINE_NUMBER_TYPE);
function SET_ON(line_number:LINE_NUMBER_TYPE);
function USED_ON(line_number:LINE_NUMBER_TYPE);
function IN_FOR_LOOP_ON(line_number:LINE_NUMBER_TYPE);
function PARAMETER_ON(line_number:LINE_NUMBER_TYPE);
function IN_WITH_ON(line_number:LINE_NUMBER_TYPE);
function CALLED_ON(line_number:LINE_NUMBER_TYPE);
function IN_GOTO(line_number:LINE_NUMBER_TYPE);
function Print_CNTRL(character);

end CROSS_REFERENCER;
```

The interface to the symbol table has three main sets of functions:

1. **Object Management functions:** which provide facilities for creating or finding objects in the symbol table guided by a set of visibility rules.
2. **Scope Management functions:** which manage the scopes in a program and regulate the visibility of objects.
3. **Object Usage functions:** which provide facilities for recording objects usage information (defined on, set on, used on) these are generated by meta tool.

The object management functions are the following:

1. **function** CONNECT_TO_OBJECT(Identifier:CHARACTER_STRING_TYPE);
2. **function** DISCONNECT_FROM_OBJECT();

The CONNECT_TO_OBJECT function locates an object in the symbol table or if it does not already exist it creates an object. Once an object has been located all information sent to the symbol table is sent to this object until the DISCONNECT_FROM_OBJECT function is activated by the parser.

The scope management functions are the following:

1. **function** ENTER_SCOPE(). The ENTER_SCOPE facility instructs the symbol table that the parser has identified a new scope.

2. **function** USE_SCOPE(Identifier:CHARACTER_STRING_TYPE). The USE_SCOPE function adds the scope with the name stored in the parameter to the list of currently visible scopes.

3. **function** FINISHED_WITH_SCOPE(). This function removes the scope that was added to the list of visible scopes.

4. **function** EXIT_CURRENT_SCOPE(). The function removes the scope that was last added to the list of visible scopes using the ENTER_SCOPE facility.

5. **function** MAKE_OBJECT_PUBLIC(Identifier:CHARACTER_STRING_TYPE). This function labels an object as being visible to scopes that use this scope.

6. **function** MAKE_OBJECT_PRIVATE(Identifier:CHARACTER_STRING_TYPE). This function labels an object as being not visible to scopes that use this scope.

The object usage functions are the following:

1. **function** PARAMETER_CLASS();

2. **function** LABEL_CLASS();

3. **function** VAR_CLASS();

4. **function** CONST_CLASS();

5. **function** TYPE_CLASS();

6. **function** FUNCTION_CLASS();

7. **function** PROCEDURE_CLASS();

8. **function** NAMESPACE(Source_file_name:CHARACTER_STRING_TYPE);

9. **function** VAR_AS_PARAMETER(line_number:LINE_NUMBER_TYPE);

10. **function** CLASS();

11. **function** PASCAL_TYPE();

12. **function** FUNCTION_TYPE(Identifier:CHARACTER_STRING_TYPE);

13. **function** EXTERNAL_ON(line_number:LINE_NUMBER_TYPE);

14. **function** FORWARD_DECLARED_ON(line_number:LINE_NUMBER_TYPE);

15. **function** CONSTANT_VALUE();

16. **function** NON_LOCAL(line_number:LINE_NUMBER_TYPE);

17. **function** DECLARED_ON(line_number:LINE_NUMBER_TYPE);

18. **function** SET_ON(line_number:LINE_NUMBER_TYPE);

19. **function** USED_ON(line_number:LINE_NUMBER_TYPE);

20. **function** IN_FOR_LOOP_ON(line_number:LINE_NUMBER_TYPE);

21. **function** PARAMETER_ON(line_number:LINE_NUMBER_TYPE);

22. **function** IN_WITH_ON(line_number:LINE_NUMBER_TYPE);

23. **function** CALLED_ON(line_number:LINE_NUMBER_TYPE);

24. **function** IN_GOTO(line_number:LINE_NUMBER_TYPE);

The function names which end in _CLASS , such as VAR_CLASS are activated when a new class of object is encountered. For example when parsing the language Pascal and a VAR declaration is detected, the VAR_CLASS function should be activated informing the cross referencer tool that the next declaration will be of the class VAR. The function CLASS records the current class in the object that has been connected to with the CONNECT_TO_OBJECT function. The remaining functions described will record the use of objects. For example, if an object is declared the following functions should be activated:

1. CONNECT_TO_OBJECT(Identifier);

2. DECLARED_ON(line_number);

3. DISCONNECT_FROM_OBJECT();

Another important feature of this system is the **function ADD_TO_TEXT_BUFFER** which is a facility to add the current token, taken from the parser to a text buffer. This facility is useful as it can be used to place a sequence of tokens into an objects slot. For example when cross referencing this Pascal declaration:

```
Var X: array [1..10] of real;
```

the tokens 'array [1..10] of real' will have to be stored in a buffer and when the parser has detected the end of the type definition for the variable X the tokens stored will have to be stored in the object associated with the identifier X.

One of the main objectives other than making the model language independent is to simplify the symbol table routines, for example, looking up and defining objects. This particular model has a function called CONNECT_TO_OBJECT which locates an object associated with identifier passed to the symbol table, or it will automatically create an object of the structure of the current class of symbol.

Visibility Rules

Visibility is the main criterion for finding an object associated with an identifier. Each object belongs to a particular class of identifiers or symbols which it describes. The class of the object determines the properties of the object, for example, the object usage or its location in the symbol table. The rules for locating objects in a symbol table are determined by considering the class of a particular object and also the visibility assumptions built into the cross referencer tool. The process of finding an object associated with an identifier found by the parser should be deterministic, although during the process of finding an object there may be several possible candidates. This is because a program may have many contexts or scopes in which identifiers with the same name can exist. Therefore the visible objects must be analysed in order to determine which object is visible from the point where it was referenced in the source code. The resolution process eliminates candidates by consulting the visibility rules and the particular object classes. This process can be

quite complex as the visibility differs for different classes of objects.

It was decided that this particular model would make many assumptions about visibility rules and has two main advantages:

1. The specification is kept concise and easy to understand.
2. The specification will also be non procedural.

The main part of the model to deal with visibility is a scope stack which will contain a series of pointers to symbol tables. Each scope has a symbol table and when the parser enters a scope this scope is pushed on to the stack using the function ENTER_SCOPE. Therefore at any given time the scope stack will contain the set of objects that are visible from a particular point. To find an object associated with an identifier the function CONNECT_TO_OBJECT is used and this function searches the scope stack top down. The first object found is assumed to be the object visible from that particular point in the program where it was referenced. Therefore the CONNECT_TO_OBJECT function will search the rest of the scopes until an object can be associated with the identifier. There will be only one object visible or none at all. Associating an object with an identifier is more easily understood if separated into its constituent phases.

1. The identifier to be looked up, the identifier's class and the identifier's expected type are passed to the symbol table for lookup.
2. The only editing of identifiers is case conversion and this can be switched on and off as required.
3. The scope stack contains the group of objects which are visible from the point where the look up function CONNECT_TO_OBJECT() was activated.
4. These scopes will contain all the possible object candidates which can be associated with an identifier.
5. The searching follows the path of nesting except, when a module has been included, from the innermost scope to the outer program main scope.
6. If one or more objects are found in the same scope they can be forward declared or overloaded.

7. Some objects will be ignored if not publicly visible.

8. Objects can be made visible to other scopes by exporting them. The import facility would add the named object to the symbol table associated with the scope, where the import facility was requested.

Automatic Creation of Objects

If an object is not found then this reference to the identifier must therefore be the first and CONNECT_TO_OBJECT automatically creates an object of the current class in the current scope. The assumptions made, concerning object definition are the following:

1. The new definition can hide old ones.

2. The new definition can be an overloading of old ones. Before any object is looked up by the symbol processor, the parser sends a message to the symbol table defining the class of object containing the identifier and also the expected type of the identifier. This aids the cross referencer to locate the correct object.

3. If a forward definition is discovered the second occurrence of the identifier is moved to the first occurrence, replacing it and making it visible at the point where it is forward declared.

As the source code is assumed to be compilable then conflicting definitions should be resolved using these three assumptions. Another feature supported by the model is the type of construct such as the Pascal language's WITH statement. This feature is used for nested data structures and is very similar to the way nested scopes are handled, using a scope stack. Other assumptions are made about scopes as it was discovered in the survey that there are many different types of scopes in programming languages.

As the model assumes that source code being cross referenced is compilable, then Ada overloading will only be found when processing Ada and the WITH statement will only be found when parsing Pascal. Therefore it is not necessary to include in the cross referencer specification a procedural description of what to do. This makes the specification much simpler than the approach

taken by [32,33]. This model is a very simple model designed specifically for the problem of producing language independent static analysis tools and does not provide facilities for binding internal storage to objects.

An Abstract Cross Reference Listing Printer

One of the original objectives of this project was to capture the basic cross referencer algorithm into an abstract form, to make the tool a more versatile tool. The most desirable form of coupling between modules is a combination of stamp and data coupling and the most desirable form of cohesion is informational [12].

A cross referencer produces a list of objects and their properties. The intermediate cross reference listing printer could be separated from the other components resulting in a printer which is not bound to the symbol table by global data structures like CXR and PXR [6,10]. This was the approach adopted during the design of the printer for the generated cross referencer.

There are two main tools used in printing a cross reference listing:

1. A **Printer:** for emptying the contents of cross reference symbol table into a text file in an intermediate representation. This is a machine readable file with printer control characters embedded into it.
2. A **Formatter:** for scanning this intermediate listing of cross reference information and producing the formatted readable listing.

This project focuses on the **Printer** tool for producing an intermediate representation. This printer tool is part of the cross referencer and would be automatically activated.

The method of printing would be the following:

1. The user would select the style of listing required, structured or alphabetic, by requesting this on the command line when the cross referencer was executed.
2. The objects and their property lists would be printed in the particular order specified, either structured or alphabetic lists.

The call from the parser to the intermediate listing printer, to print the listing, would be made with a single procedure call. The lower level details will be left to the printer.

One type of listing will be the machine readable listing which will be stored in a file for subsequent use by another tool. The other listing will be a readable formatted listing sent to standard output. Each object slot value will be preceded by a meta slot value, that is, a piece of text describing the meaning of the slot value. This can be customised by the tool implementor when generating the cross reference tool by a merger tool.

One of the primary functions of the formatting tool is to format the cross reference listing by inserting spaces and linefeeds between the tokens and also to break up the lines which are too long.

Two approaches to formatting the listing were investigated. One approach would be to parse the source code file to and produce a parse tree. A set of routines called by the parser could insert line breaks and indentation into the parse tree as it was being constructed. This approach would be syntax directed and a separate tool could be employed. The printer would have a knowledge of the constructs of the cross reference listing and also any punctuation. The specification would be a context free grammar which would include information about the spaces, indentation and linefeeds. The other approach would be to insert formatting commands into the object property lists in the symbol table as the objects were created. For example:

CONTROL CHARACTER	FUNCTION
n	newline
i	indent if not page overflow
s	set tab to previous tab
m	set indentation to margin
b	insert blank line

When the printer is printing the object property list for an object it checks for control characters and when found performs a formatting option. The second approach is more efficient than the first as less parsing would have to be performed in order to produce a cross reference listing from a piece of source code. This is important when dealing with industrial scale software as a detailed cross

referencer listing can be twice as long as the source code. In the event of a new type of object being added to the cross referencer, the cross reference listing produced would now have a new structure to accommodate this new object. Therefore the parser would have to be modified. The control characters approach was adopted as it is the more efficient.

If the maintenance programmer is using a printer tool it may be convenient to print the listing on a printer. It would be quite possible to construct a printer that includes text processor commands in its output. So commands to bold face key words and to italicize object names could be inserted into the cross reference listing. This would improve the readability of the cross reference listing. It may also be possible to print paginated listings complete with line numbers, page numbers, page titles and numbers, and an index of objects.

If the maintenance programmer is using a windows environment with a bit mapped screen, the formatter tool can be made even more efficient by only formatting those lines that fall within the window.

To summarize the formatter takes an intermediate representation of the cross reference listing produced by the printer tool which can be alphabetic or structured, then it prints each object property list, and the contents of all the slots associated with each object. Each slot in the intermediate listing is traversed left to right and top down and the formatter functions are called each time a control character is discovered. A control character is indicated by a back slash followed by a letter. There is also a plug in file containing pairs of records, the first of the pair is the control character and the second record is what is to be printed. By using this very simple file the formatter can also be customised easily without doing any coding.

A routine called **Print_CNTRL**(character); inserted in the parser grammar will insert the control character passed, into the object which is currently connected and is written in the slot value that was written to last. More than one control character can be inserted into a slot forming a command list of formatter control characters which will be encapsulated in square brackets. The square brackets will facilitate easy detection of groups of control characters in the intermediate cross reference listing by the formatter tool.

Defining the Specification Input

The approach taken by Reiss[32] and by Koskimies, Nurmi and Paakki[21] is to present all the details at once. This toolkit uses a top down approach which makes it easier for the tool implementor to describe his symbol table, however it becomes slightly harder to transform the specification into some sort of intermediate representation. A top down approach was used when designing the notation for describing the cross referencer tool, rather than a flat specification. The specification consists of nine sections:

1. **Object classes**
2. **Data items passed from the parser**
3. **Semantic Routines**
4. **Semantic Routine Usage**
5. **Text Buffer Usage**
6. **Cross Referencer Object Structures**
7. **Object Slot Descriptions**
8. **Source Code Identifiers to be ignored**
9. **Listing Specification**

The **Object classes** section describes the different types of object to be stored in the symbol table. An example of an object class description is given below:

```
<procedure_class>
selected_by_routine procedure_class();
intermediate_rep_name ".9procedure";
when_same forward_def;
```

When an object is declared it must be stored according to its class. Therefore a function must be used to select the appropriate class. The purpose of the "selected_by_routine" statement in the

cross referencer specification is to choose the right class for a particular object. All objects will appear in the intermediate cross reference listing and will have an intermediate name. The clause in the specification "intermediate_rep_name" describes this intermediate name. Therefore when a parameter is referred to in the cross reference listing it will be referred to by its intermediate name. Another clause which can be used when specifying objects is the "when_same forward_def" statement. This instructs the cross referencer that when an object is is declared more than once in the same scope, it is to be forward declared.

The **Data items passed from the parser** section describes the type of data items passed from the parser. The cross referencer model currently has three data types defined as:

```
DATA_ITEMS_PASSED_FROM_PARSER;
<char *> character_string;
<char *> source_file_name;
<int>    line_number;
end;
```

The **Semantic Routines** section describes all the routines that record the use of objects in a source program. They are numbered so they can be quickly referenced further on in the specification. Each function has a name and the type of data item that is passed to it.

```
SEMANTIC_ROUTINES;

[1]  object(character_string);
[2]  namespace(source_file_name);
```

The **Semantic Routine Usage** section describes which of the semantic routines are used by the various objects in a program. The semantic routines used are listed for each object.

```
<var_class>   uses_semantic_routines  [1,2,3,4,5,11,12,14,16]
```

The **Text Buffer Usage** section describes lists the semantic routines which take text from the text buffer and place it in an objects slot structure. For example:

```
TEXT_BUFFER_USAGE;

const_value();
pascal_type();
end;
```

The **Cross Referencer Object Structures** section describes the list of slots that each object structure has. For example:

```
<var_class>  slot_structure_is
(object,
namespace,
class,
pascal_type,
declared,
used
var_as_parameter
inwith
infor)
```

The slots have the same names as the semantic routines. This is because for example the function called will deposit a data item of type linenumber in a slot with the name 'called'.

The **Object Slot Descriptions** section describe the characteristics of each slot in an object. For example:

```
set :
type_id          is line_number
type             is int
slot_value       is multi
meta_slot_value is +04set
```

A slot has a type identifier and actual type. A slot can be single or multiple value. Therefore a slot could contain one or more line numbers. The name describing it in the intermediate cross reference listing is also described as the meta_slot_value.

The **Source Code Identifiers to be ignored** section describes any reserved words which should not be stored in the cross reference listing if any.

The **Listing Specification** section describes the slots which are to be included in a particular level of cross reference listing. There are three levels of detail full, intermediate and terse. The tool implementor can specify the various slots to be included and the printer will filter out the rest. For example the terse listing specification:

```
TERSE_SPEC;
object;
namespace;
class;
declared;
set;
used;
end;
```

The appendix D has the lexical analyser specification for this notation and appendix E contains the grammar of the cross referencer specification. The appendix C contains a cross referencer specification for the programming language Pascal.

Often generator specifications allow user code called 'escapes' to be supplied to add extra functionality to the program generated[9]. This facility was not included in order to maintain a

non procedural high level solution.

Designing a code generating method

Since much of the variant code in a cross referencer has a pattern, the specification of it can be mechanically translated into program source code. The generator can be guided by using a template and by extracting the necessary information from the specification of the cross referencer. The code generator tool will be guided like any other tool which uses a template, such as a tool which uses a guide for cutting or drilling material.

Many parts of the template will need to be replicated, therefore some mechanism for iterating the code patterns is necessary. The template will consist of an implementation code and indication of the text to be inserted into this. The location of the text insertion could be marked by a cell i.e. the source of the text to be inserted enclosed in angular brackets.

The generator will essentially be a macro substitution tool which will emit compilable code. The generator will be a series of small subtools which will initially form an experimental toolkit for producing a tool for automating the production of front ends.

The template approach will allow the tool implementor to alter the template. This would facilitate easier modification of the code and also it may allow the underlying implementation language to be changed since the text substitution tool is language independent.

This type of generator is of an experimental character. The generating method was determined by looking at the application domain. The application domain was designed by looking at the requirements provided by the domain analysis i.e. chapter 4 Language Features to be Cross Referenced.

Code templates

A code template is a text file of program code which contains areas designated for textual substitution. This inserted text comes from the textual specification of the cross referencer. The code templates will be set up each time the language emitted from the substitution tool needs to be changed to another language. However it is unlikely that the underlying language of the cross referencer tool would change to another language so the templates will not need to be altered frequently. The templates are essentially program fragments with areas designated for textual substitution. The text is taken from the cross referencer specification.

Appendix F contains a code template for the cross referencer symbol table. Each template has an identifier as the first token "macrolist" followed by a number which represents the number of the subtool which uses this template. The templates constructed consist of program code written in the programming language C. Areas for textual substitution are designated with the use of angular brackets called cells. The following is an example:

```
struct <<SAME>> <OBJECT_CLASSES>_section
{
struct <<SAME>> <OBJECT_CLASSES>_object  * <<SAME>> <OBJECT_CLASSES>;
struct <<SAME>> <OBJECT_CLASSES>_section * left;
struct <<SAME>> <OBJECT_CLASSES>_section * right;
};
```

The cell indicates to the textual substitution tool where to obtain the text from. There are twelve possible sources of text. These are the cross reference product description files. These are extracted from the cross referencer specification and the substitution tool merges this text with the templates to produce a compilable program. So each cell is essentially a the name of a file from which text is to be taken. Each cell is preceded by a cell with double angular brackets. The purpose of this cell is to indicate whether to take the next piece of text from the product description files or whether to use the same piece of text that was last read from the product description files. These cells are either NEXT or SAME respectively.

The Intermediate Files

The product description files are an intermediate representation of the cross referencer tool specification. They are summarised below:

1. object_classes: a list of object classes
2. intermediate_names:a list of names that will be used in the cross reference listing
3. type_identifiers: the identifiers for the data types of the items passed from the parser to the symbol table.
4. routine_usage: a list of routines used by each class
5. object_structure: a list of object structures
6. terse_spec: list of slots to be included in the cross reference listing of the terse level of detail
7. class_selection: the routines to indicate the class of an object
8. data_types: a list of the data types used in the object structures
9. semantic_routines: a list of the semantic routines
10. text_buffer_usage: a list of the routines that use the text buffer
11. slot_types: a list of the slot types
12. intermediate_spec: list of slots to be included in the cross reference listing of the intermediate level of detail

Toolkit

A single generator tool could be constructed and its user interface may look like the following:

```
$ gen pascal.xref
```

```
Cross Referencer Generator (Version 0.1)

  Phase 1 Parsing  Cross Referencer Specification
          End of file encountered line: 346
          Total Errors (0)
          Intermediate Files Created

  Phase 2 Generating Cross Referencer Object Structures
          Object Structures Created

  Phase 3 Generating Object Manipulation Code
          Object Manipulation Code Generated

  Phase 4 Generating Tool Interface
          Interface Created in file XRGen_Interface

  Phase 5 Linking Generated Code with Tool core
          Linked
          Cross Referencer in file xref.c

Done
$
```

The above user interface was added to the generator tool.

1. **Phase 1:** This phase analyses the cross referencer specification and produces intermediate files which will be used by the generator to produce the cross referencer.

2. **Phase 2:** This phase generates a symbol table.

3. **Phase 3:** This phase generates the code for recording object usage and for printing the contents of the symbol table as an intermediate cross reference listing.

4. **Phase 4:** This phase generates an interface to the cross reference tool

5. **Phase 5:** This phase links the generated code with the cross reference code.

It was decided that the general approach would be to build a series of subtools rather than one complex tool. If each variant aspect of the cross referencer could be addressed with a single tool this would make the design and construction much simpler than having one complex tool.

Code Generation Process

The following diagrams overleaf, show the proposed cross referencer tool generation process. The process consists of three phases:

1. **Analysing the cross referencer specification**

2. **Generating the tool components**

3. **Integrating the tool components into a single tool**

The first phase shown in fig 2 is the Cross Referencer Specification Analysis made by parsing the cross referencer specification, and producing an intermediate representation of the specification which can be processed by the code generating tools.

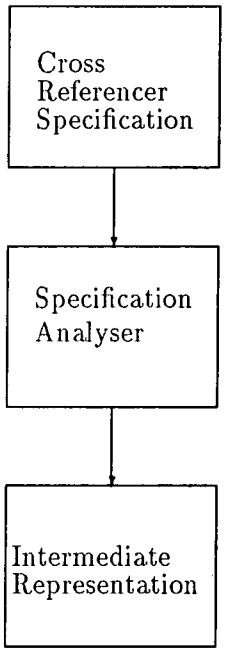

fig. 2 SPECIFICATION ANALYSIS

The second phase shown in fig. 3, Code Generation, includes the use of three subtools to produce the cross referencer variant components, the Semantic Routines, the Printer Routines and the Symbol Table Structure itself. Subtool one is the tool that produces the symbol table for the cross referencer. Subtool two is the tool for producing the semantic routines for storing static information into the symbol table and subtool three is the tool for generating the variant parts of the printer.

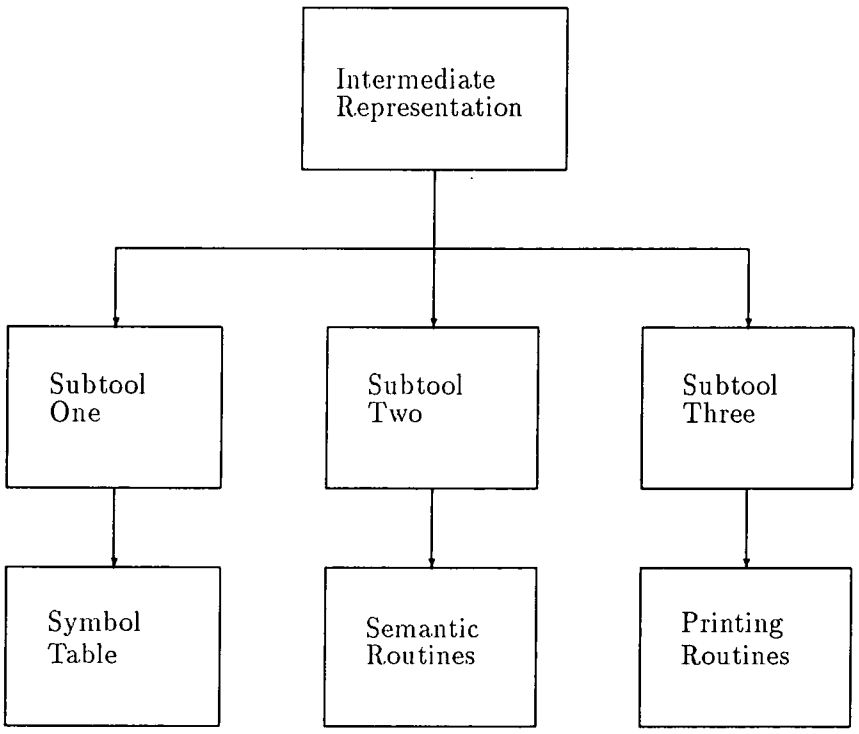

fig. 3 CODE GENERATION

The final phase shown in fig. 4, connects the variant and the invariant code, the language independent portion of the tool by compiling these files with compiler to produce one executable object code file. The single object code file produced is the Cross Referencer.

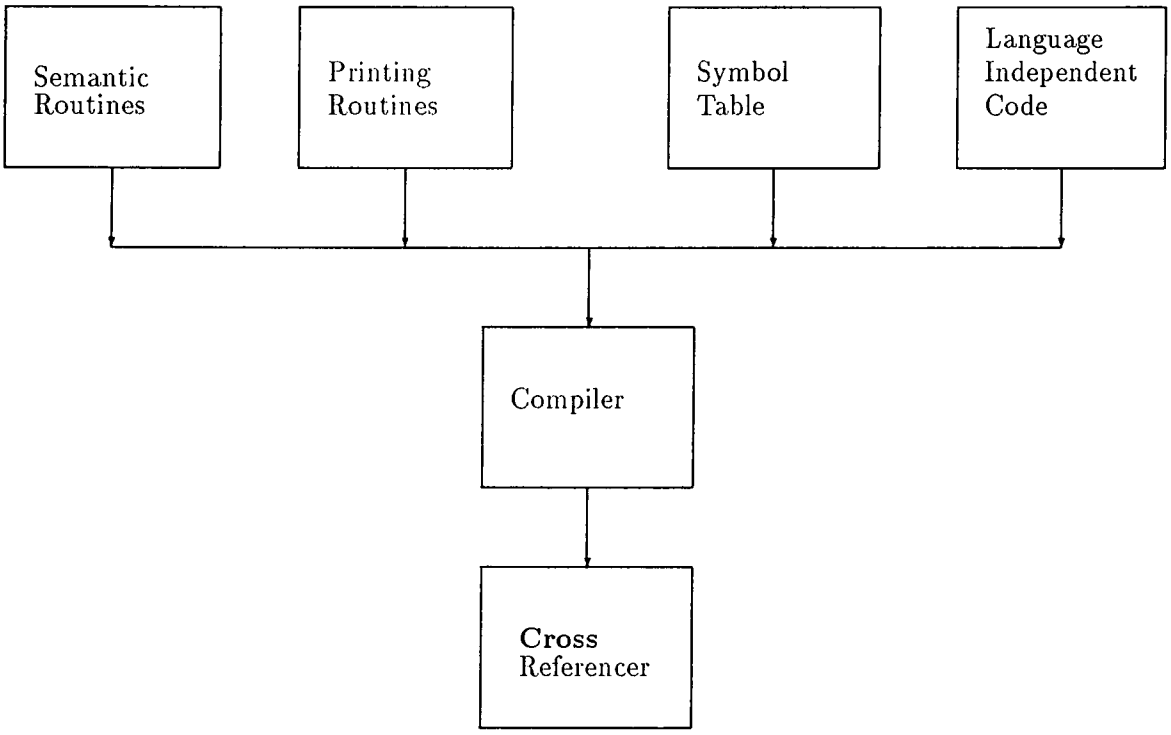

fig. 4 TOOL CONNECTION

Generating a Cross Referencer

A code generation tool would re-use the language independent parts of the cross referencer and customise the variant parts of the cross referencer using the cross referencer model as a basis for doing this. The generation process from specification to usable tool would consist of the following:

1. Produce Lexical Analyser Specification
2. Generate Lexical Analyser
3. Produce Parser Specification
4. Generate Parser
5. Debug the Parser
6. Produce Cross Referencer Specification
7. Generate Intermediate Specification Files
8. Generate Symbol Table Data Structure Code
9. Generate Printer Code
10. Generate Semantic Routines
11. Add language Independent Code
12. Compile Cross Reference Tool

Meta Tools constructed

Meta tools are tools for generating other software tools. This type of tool was used during stages seven to ten above. The four main tools constructed are:

1. **Cross Referencer Specification Analyser Tool**
2. **Component Template Preprocessor Tool**

3. **Intermediate File Modifier Tool**

4. **Text Substitution Tool**

The function of the **Cross Reference Specification Analyser Tool** is to parse a text file containing the cross reference specification notation described earlier. See appendix E for the grammar of the cross reference specification. During the parsing of the specification routines are called by the parser to store pieces of text in the text files called intermediate files. Twelve intermediate files are produced for subsequent use by the text substitution tool. These intermediate files contain text to be inserted into the component templates. This tool was written using YACC and the routines to create the intermediate files were written in the programming language C. For example the following is an extract from the YACC specification of specification analyser. It contains the grammar for the specification for an object structure and the actions which produce the intermediate files.

```
structure_definitions
    : structure_definition
    | structure_definitions structure_definition
    ;
structure_definition
    : '<' IDENTIFIER_TOKEN   {store_in_object_structure(yytext);}
    | '>' SLOT_STRUCTURE_IS_TOKEN
      '('                    {store_in_object_structure(yytext);}
       slots ')'             {store_in_object_structure(yytext);}
    ;
```

The function store_in_object_structure(yytext) sends the text which is the current token at that instant, to an intermediate file called object structures. The YACC specification for this tool can be seen in appendix E. It was a general strategy to try an implement these meta tools using the YACC compiler compiler as it will mean that the meta tool can be changed by changing its specification rather than the underlying source code.

The function of the **Component Template Preprocessor Tool** is to modify the templates used by the text substitution tool. For example a template may represent a record structure, however the number of fields may not be known until the specification has been parsed. So if there were seven fields required in the record structure then the field part of the template would be replicated seven times. The purpose of this tool is to modify the parts of the template that can not be known until a cross reference specification has been parsed. Appendix F shows a symbol table template. The template fragment below is one such part of the template that must be replicated. It is the skeleton for a binary tree of objects. Each class of objects must have a structure like the following:

```
struct <<SAME>> <OBJECT_CLASSES>_section
{
struct <<SAME>> <OBJECT_CLASSES>_object  * <<SAME>> <OBJECT_CLASSES>;
struct <<SAME>> <OBJECT_CLASSES>_section * left;
struct <<SAME>> <OBJECT_CLASSES>_section * right;
};
```

If a programming language had two classes of identifier then this part of the template must appear twice.

The function of the **Intermediate File Modifier Tool** is to modify some of the intermediate files produced by the specification analyser. The notation for describing the cross referencer requirements was designed for the tool implementor rather than designing it for easy analysis by a tool. Consequently when some text is is extracted from the specification it is not immediately in a form suitable for merging with component templates. Therefore some intermediate file modifier tools were invented. For example one intermediate file contains a list of variable identifiers and their actual type in the programming language C:

```
character_string  char*
linenumber        int
```

As the program code emitted from the generator is in the language C the type must come before the variable name. This tool swaps the order of text .

The function of the **Text Substitution Tool** is to merge text from the intermediate files with a component template. It is a simple tool constructed with the Unix tool YACC. This grammar for the templates is described as a YACC specification and the program was generated using the compiler compiler YACC. Actions were also written and inserted into the YACC grammar. These routines will take tokens in the template and text from the intermediate files and write them to the file containing the compilable tool components written in C. This tool is the same in design and the way it operates as the tool that analyses the cross referencer specification and produces the intermediate files(product description files).

Connecting the Parser to Symbol Processor

The functions in the interface file to the symbol processor should be inserted into the programming language parser YACC grammar. Also, items passed from the parser to the cross referencer usually originate from the lexer. These data items should be made visible in the parser specification so they can be passed as parameters to the cross referencer. For example the current line number, the source file being cross referenced and the current token. Few have attempted to automate this aspect, perhaps because little research has been done in generating symbol processing mechanisms from specifications.

A Tool to Connect the Parts

Two possible ways of reducing the effort to connect these two parts of the tool:

1. A tool which automatically inserts the function calls, guided by a specification.
2. The use of a windows environment with a tool to transfer text from one window to another.

This is an example of a parser grammar rule without any symbol processing functions embedded in it. It is written to parse a Pascal Var declaration such as *Var x,y,z: integer;* :

```
variable_declaration_part
        : VAR_TOKEN  variable_identifier_part ';'
        ;
variable_identifier_part
        : identifier_list ':' type
        | variable_identifier_part ';' identifier_list ':' type
        ;
identifier_list
        : IDENTIFIER_TOKEN
        | identifier_list ',' IDENTIFIER_TOKEN
        ;
```

The same example with the symbol processing functions embedded in the grammar is:

```
variable_declaration_part
        : VAR_TOKEN              {VAR_CLASS();}
          variable_identifier_part ';'
        ;
variable_identifier_part
        : identifier_list ':'
          type                   {TYPE_CLASS();}
        | variable_identifier_part
          ';' identifier_list ':'
          type                   {TYPE_CLASS();}
        ;
identifier_list
        : IDENTIFIER_TOKEN       {CONNECT_TO_OBJECT(Identifier);
                                  NAMESPACE(source_file_name);
                                  DECLARED_ON(line_number);
                                  DISCONNECT_FROM_OBJECT();}
```

```
| identifier_list  ','

  IDENTIFIER_TOKEN      {CONNECT_TO_OBJECT(Identifier);
                         NAMESPACE(source_file_name);
                         DECLARED_ON(line_number);
                         DISCONNECT_FROM_OBJECT();}

;
```

The VAR_CLASS function would be called when the VAR_TOKEN is detected by the parser. This will inform the symbol table that the current next object to be inserted into the symbol table will be of VAR class. As an IDENTIFIER is detected by the parser the object is created or located in the symbol table with the CONNECT_TO_OBJECT function. The name of the source file the Identifier is resident in is stored using the NAMESPACE function and the line it is declared on is recorded in the symbol table with the DECLARED_ON function.

It is technically feasible to write an automated tool to insert these symbol processing functions guided by a specification, which may contain the rule name, the token name and the commands to be inserted. However in creating this specification, text from the cross referencer interface has been transported to this specification file, therefore it might as well be transferred straight to the parser.

5.4 An Example

An example of the use of the toolkit is described in this section. The toolkit was used to generate compilable program code written in the programming C. The programming language Pascal was used as an example language on which a cross referencer may operate. The following is described:

1. A Pascal Cross Referencer Specification
2. The Production of Intermediate Files

3. The Modification of the Intermediate Files

4. The Use of the Text Substitution Tool

5. The Generated Symbol Table Data Structure

A Pascal Cross Referencer Specification was created using the notation described earlier. This can be seen in appendix C. The specification has nine sections whose content are specific to the programming language Pascal. For example the first section contains the specification for the classes of objects that will represent the different types of identifier found in Pascal. A fragment of this section is shown below:

```
OBJECT_CLASSES;

     <parameter_class>
          selected_by_routine parameter_class();
          intermediate_rep_name ".0parameter";
     <label_class>
          selected_by_routine label_class();
          intermediate_rep_name ".4label";
     <var_class>
          selected_by_routine var_class();
          intermediate_rep_name ".7variable";
```

The identifier in the angular brackets represents the name of the identifier. The selected_by_routine clause indicates how the parser will inform the symbol table which type of object it is processing and the intermediate_rep_name clause indicates the name to be given to this object in the intermediate cross reference listing produced. There are seven types of identifiers described.

The next section describes the text items which are to be transferred from the source code to the symbol table. These data items are used as parameters in the semantic routines described earlier. For example :

```
DATA_ITEMS_PASSED_FROM_PARSER;

     <char *>  character_string;
     <char *>  source_file_name;
     <int>     line_number;
     end;
```

The next section in the Pascal cross referencer specification is the description of the semantic routines. These all have the same function, that is, they insert text into the object slot structures. Their names are the same as the names of object slot structures and the parameters used are also identified. For example the fragment below shows how text will be mapped to object slots whose names are object, namespace and var_as_parameter:

```
SEMANTIC_ROUTINES;

   [1]  object(character_string);
   [2]  namespace(source_file_name);
   [3]  var_as_parameter(line_number);
```

The numbers in square brackets are a means of quickly referencing this function in other places in the specification. For example the next section in the cross reference specification represents the use of the semantic routines. Each object will use specific routines depending on its slot structure. For example:

```
SEMANTIC_ROUTINE_USAGE;

   <parameter_class> uses_semantic_routines  [1,2,4,12,15,16]
   <label_class>     uses_semantic_routines  [1,2,4,11,18,19]
   <var_class>       uses_semantic_routines  [1,2,3,4,5,11,12,14,16]
```

Therefore the object class parameter_class will use the semantic routines whose numbers are 1,2,4,12,15,16. The semantic routines which use the text buffer are indicated in the next section of the cross reference specification by listing the semantic routine names. The next major section of the cross reference specification is the object slot structures. For example the object structure for a var class object is given below.

```
CROSS_REFERENCER_OBJECT_STRUCTURES;

        <var_class> slot_structure_is
        (object,
        namespace,
        class,
        pascal_type,
        declared,
        used,
        var_as_parameter,
        inwith,
        infor)
```

The name of the object class is given in angular brackets. The identifiers enclosed by round brackets are the names of slots. These slot names can be thought as being similar to field identifiers in the record construct.

The next section describes the actual slot itself. Each slot has a type which can be character_string and each slot holding line numbers can be a multiple value slot. Also each slot has a meta slot value which is the identifier representing this slot in the cross reference listing. An example of the used slot is given below:

```
used :
        type_id         is   line_number
        type            is   int
```

```
slot_value      is   multi
meta_slot_value is   +05used
end used
```

The next section of the cross reference specification is the list of identifiers whose use is not to be recorded in the symbol table. For example in the language Pascal the function "abs" will be detected as a function identifier although it is not to be recorded in the symbol table. The final section of the cross referencer specification is the cross referencer listing specification. Each language dependent implementation will have certain slots that will only be needed by that particular language. The tool implementor can choose which slots shall be output in the different types of intermediate cross reference information. There are three levels of detail, described earlier. An example of this part of the specification is shown below:

```
LISTING_SPECIFICATION;  /* Object packets to be included in listing */

     TERSE_SPEC;
         object;
         namespace;
         class;
         declared;
         set;
         used;
         end;

    INTERMEDIATE_SPEC;
         object;
         namespace;
         class;
         declared;
         set;
         used;
         undefined;
```

```
        inwith;
        infor;
        ingoto;
        end;

end; /* listing specification */
```

The Production of Intermediate Files follows after the specification has been produced. The specification analyser produces these files. This tool will also detect any syntax errors in the specification. The system is not capable of producing program code until the cross referencer specification is syntax error free. **The Modification of the Intermediate Files** is performed and the new files are then automatically copied back to the originals before using them in the next phase of the tool generation. **The Text Substitution Tool** is invoked and the name of the template "macrolist1", which describes the variant symbol table structure, is passed as an argument to this tool. The symbol table component is then produced. **The Generated Symbol Table Data Structure** is written in the programming language C, shown in appendix G. It consists of pointers to data structures which represent program scope sections.

```
struct scope
{
struct  parameter_class_section *  parameter_class_ptr ;
struct  label_class_section *  label_class_ptr ;
struct  constant_class_section *  constant_class_ptr ;
struct  type_class_section *  type_class_ptr ;
struct  var_class_section *  var_class_ptr ;
struct  function_class_section *  function_class_ptr ;
struct  procedure_class_section *  procedure_class_ptr ;
struct scope * next;  /* order of scopes */ };
}
```

In each scope there are different classes of objects. Each struct will hold all the objects of that particular class. Each data structure for storing pointers to objects is implemented as a binary tree.

The reason for this is so that all of the objects can be printed in a structured listing by traversing this data structure and printing each element of the data structure. For example the following is one such binary tree of pointers to objects of the var class.

```
struct  var_class_section
{
struct  var_class_object *  var_class_ptr ;
struct  var_class_section * left_ptr;
struct  var_class_section * right_ptr;
};
```

Each of these pointers will point to a particular object holding symbol usage information. Each class of objects is used in different ways so the object structures are not uniform in structure. For example:

```
struct  var_class_object
{
char   object[256];
char   namespace[256];
char   class[256];
char   pascal_type[256];
struct linenumbers *  declared;
struct linenumbers *  used;
struct linenumbers *  var_as_parameter;
struct linenumbers *  inwith;
struct linenumbers *  infor;
};

struct  function_class_object
{
char   object[256];
```

```
char    namespace[256];
char    class[256];
struct linenumbers *  parameter_on;
struct linenumbers *  external;
struct linenumbers *  forward;
char    non_local[256];
struct linenumbers *  called;
};
```

5.5 Summary

The language independent cross referencer model produced was described and a method of customising and making maximum re-use of this model was investigated. Meta tools were constructed to illustrate this particular idea. An increase in the productivity of building static analysis front end tools can be gained through customised re-usable software.

Chapter 6

Conclusion

6.1 Project Description

Software Maintenance has been identified as the most expensive phase of the software life cycle and will continue to devour resources even if formal methods of software development are widely used. Research is needed into development of tools for use by the maintenance programmer. It can be envisaged that static program analysis tools such as cross referencers, control flow and data flow analysers will be developed together as a complete package to facilitate program understanding.

The main problem perceived with these tools is that they will currently only operate on software written in one language. Software systems are written in many different languages. Many tool vendors have reflected on the idea of producing language independent tools but few have attempted to produce language independent front end tools. The proposal was a one year research activity to produce language independent front ends for static analysis tools.

6.2 Review of Major Points in each chapter

Chapter 1 **Software Maintenance:** This chapter surveys the maintenance activities and tools which can be used to support these activities. The most expensive phase of maintenance is surveyed in more detail and the problem of trying to understand how programs operate is described. Models of program comprehension are identified and the potential use of cross reference tools to aid the maintenance programmer construct hypotheses about the workings of a program are investigated. Tools to facilitate program comprehension were analysed and a research objective was established to investigate a new cross reference tool front end which can be used to process any language.

Chapter 2 **Survey of Cross Reference Tools:** The strengths and weaknesses of commercially available tools were analysed and the state of the art in cross reference tools was identified. The ideal cross referencer was also described. The British Telecom Research Laboratories and the Centre for Software Maintenance cross reference tools were used and evaluated and the difficulties facing cross referencers were identified. The external interfaces and the construction of cross reference tools were also analysed .

Chapter 3 **Cross Referencing Language Features:** This chapter discusses particular features which are present in programming languages which are relevant to cross reference tools. The symbol processing functions needed was analysed and the common features in programming languages were identified as a possible language independent skeleton. A method of reducing the amount of programmer effort to add other tool components to this skeleton was examined and a program generator was identified as a possible solution.

Chapter 4 **Program Generators:** This chapter surveys the major benefits and applications of generators. The process of generating a program using a generator is described and the advantages and disadvantages of using generators was made distinct. Two approaches to developing generators were evaluated. The literature on parsing theory, compiler techniques, compiler engineering and language processor front ends was surveyed.

Chapter 5 **The Design of a Front End Generator for Cross Referencer Tools:** This chapter contains a description of a high level solution to building cross reference tools and also describes the process of using the experimental toolkit.

6.3 Achievement of Objectives Set

The following achievements were made during the project:

- The language dependent weakness of maintenance tools was identified.
- One of the most useful cost effective tools was identified, a cross referencer. A survey was conducted, which analysed the strengths and weaknesses, the state of the art and the ideal cross referencing tool.
- The need for more than just a simple text file searching tool for cross referencing source code was identified.
- Several British Telecom Research Laboratories and CSM cross reference tools were used and their construction was evaluated.
- A solution to the problem of dealing with a multifile environment was also identified.
- Cross referencer symbol table management was investigated and a set of functions were identified to transfer information from program source code into the cross referencer symbol table.
- The language independent components of a cross referencer were identified.
- A parser was constructed for the language Pascal to develop a more advanced knowledge of tool front ends.
- A survey of programming language features was conducted to identify the features that must be included in a language independent cross referencer and also to appreciate why certain features are present.
- The common features in languages such as classes of symbol, visibility rules, symbol searching mechanisms and symbol usage were identified, and the skeleton or nucleus to which other features could be added if the situation arose was identified.
- A method of reducing the effort to change features internal to the cross referencer was investigated and the use of a generator identified.
- A design decision was made that this particular generator will make assumptions concerning the common features of languages, rather than having a very complex specification input. This approach also makes the specification non-procedural.

- A specification language was designed for a generator.
- A code generation method was designed which utilizes macro substitution and code templates. This means that the language emitted from the generator can be altered by changing the template.
- Four meta tools were constructed.
- Compilable symbol table data structures were successfully generated in the programming language C.
- A solution to language independent tools was identified, designed and investigated. The solution uses a table driven syntax analyser, lexical analyser, symbol table and cross referencer which is based on common features of languages. All four components can be re-used and customised by changing the high level specifications and re-running them through the generator.
- It was observed that the use of a generator facilitates easier integration of re-usable cross referencer tool components into future implementations.
- It was observed that some kind of abstract interface was needed between the different cross referencer components in order to reduce the effort required to put them together. In order to make the idea of using program generators to re-use and customise a language independent nucleus attractive, the cross referencer components must conform to some kind of "user model" of the tool under development. Thus the user of the components is relieved of detailed coding concerns and can instead view the available components in terms of some kind of simplified model of the cross referencer under construction.
- This is one approach to language independent cross referencers and could serve as a foundation for future work.

6.4 Future Developments

Further work needs to be done on the underlying model of a cross referencer generator which would include a more detailed analysis of language features. New languages containing features such as knowledge representation will play an important part in fifth generation computing. These type

of features will need to be included in future maintenance tools. Perhaps the features of Prolog should be included in the cross referencer model.

The field of compiler construction is a specialised field in which a large amount of re-use has already occurred, for example in the customising and re-using of the finite state automaton algorithm. The front and back end division is often used as a basis for tailoring an existing compiler to a new machine. Similarly, division into further phases such as lexical analysis, parsing, error recovery, memory allocation, code generation and code optimization provides a basis for subdividing the task of writing and understanding compilers and a basis for using parts of an existing compiler in generating new compilers. The understanding of compilers gained by experience provides an accepted 'user model' of compiler construction that is of great use in simplifying the production of new compilers.

Future work could include subdivision of the symbol processing mechanism into separate functions. It might be possible to generate each individual component with a generator similar to the components of a compiler front end. Lexical analysers or parsers are generated in this manner.

When designing a cross referencer generator there are some compromises to be made. For example, if the underlying model attempts to address many programming languages it may be that the generator becomes general purpose and the tool implementor may be required to add some features to the code emitted from the generator. This would increase the effort required to produce a language independent implementation. However the generator would have a wide application domain. If the generator model is aimed at a small application domain then it may possible to produce a language independent implementation without the tool implementor adding any program code. This type of generator will generate more of the cross referencer but for only a small number of languages. A company only using two or three languages would be better using the limited domain approach. A tool vendor which produces tools for ten languages may benefit from the more general approach.

There is much scope for research in producing specification languages or any other formalisms for describing cross referencer components. This project may precipitate research into software re-use in the production of software maintenance tools. The re-use of tool components in any tool will depend on how difficult they are to modify. It is unlikely that a generic tool will be able to recognize any programming language, however it may be possible to tailor a generic design for each

language. Therefore the generated program is not a generic tool which can process many languages, but it is a processor of the language features described in the specification.

Another possible project would be to be to investigate more closely which aspects of a cross referencer may be amenable to change and produce a grammar based specification for each aspect. Tools such as an editor could be used for assisting in modifying those specifications to reconfigure the cross reference tool. This would not be complete regeneration of the whole tool. Just the components that need modification would be changed.

To summarize there are at least four approaches to language independent cross referencer tools:

UNIVERSAL: This approach requires one cross referencer to provide facilities for a wide class of programming languages. The deficiency is that it will be very large and complicated to implement.

LANGUAGE INDEPENDENT COMPONENTS : This approach requires explicit use of programming skills to integrate the components into new tools.

AUTOMATICALLY GENERATED: This approach utilizes grammar driven generators to re-use and customise a generic skeleton.

DYNAMICALLY RECONFIGURABLE: This approach intelligently modifies specific components of a language independent cross referencer without performing a complete regeneration of the cross referencer tool.

Appendix A

Pascal Lexical Analyser Specification

```
%{

/*
 *        pascal -- lexical analysis
 */

int linenumber  =1;
char loweryytext[YYLMAX];
static int screen();
%}
letter                          [a-zA-Z]
digit                           [0-9]
letter_or_digit                 [a-zA-Z_0-9]
white_space                     [ \t\n]
blank                           [ \t]
return                          [\n]
double_quote                    ["]
scale_factor                    [{digit*}]|["+"{digit}*]|["-"{digit}*]
%%

".."                                        return token(EL_TOKEN);
\"([^"\n]|\"\")+\"                          return token(STRING_TOKEN);
\'([^'\n]|\'\')+\'                          return token(STRING_TOKEN);
{digit}+"."{digit}+                         return token(REAL_TOKEN);
{digit}+"."{digit}+"E"{digit}*              return token(REAL_TOKEN);
{digit}+"."{digit}+"E""+"{digit}*           return token(REAL_TOKEN);
{digit}+"."{digit}+"E""-"{digit}*           return token(REAL_TOKEN);
```

```
{digit}+"."{digit}+"e"{digit}*                  return token(REAL_TOKEN);
{digit}+"."{digit}+"e""+"{digit}*               return token(REAL_TOKEN);
{digit}+"."{digit}+"e""-"{digit}*               return token(REAL_TOKEN);
{digit}+"e"{digit}+                             return token(REAL_TOKEN);
{digit}+"e""+"{digit}+                          return token(REAL_TOKEN);
{digit}+"e""-"{digit}+                          return token(REAL_TOKEN);
{digit}+"E"{digit}+                             return token(REAL_TOKEN);
{digit}+"E""+"{digit}+                          return token(REAL_TOKEN);
{digit}+"E""-"{digit}+                          return token(REAL_TOKEN);
[0-9]*                                          return token(INTEGER_TOKEN);
"<>"                                            return token(NE_TOKEN);
"<="                                            return token(LE_TOKEN);
">="                                            return token(GE_TOKEN);
":="                                            return token(AS_TOKEN);
"\n"                                            { linenumber = linenumber + 1; }
{return}                                        { linenumber = linenumber + 1;}
{letter}{letter_or_digit}*                      {return screen();}
{letter}({letter_or_digit}|"-"|"_")*            {return token(IDENTIFIER_TOKEN);}
{digit}+                                        { return token(CONSTANT_TOKEN); }
{blank}                                         {}
{white_space}                                   { linenumber  = linenumber + 1; }
"'"                                             { return token(QUOTE_TOKEN); }
^"#"{blank}*"include"                           { return token(INCLUDE_TOKEN);}
^"%"{blank}*"include"                           { return token(INCLUDE_TOKEN);}
.                                               { return token(yytext[0]); }
"(*""("*([^*)]|[^*]|[.]|("|"*"[^)])*"*"*"*)" {}
"{""{"*([^}]|[^*]"{"|"*"[^}])*"*"*"}"           {}


%%


/*
 *       reserved word screener
 */

static struct rwtable {          /* reserved word table */
        char * rw_name;          /* representation */
        int rw_yylex;            /* yylex() value */
        } rwtable[] = {          /* sorted */
        "and",           token(AND_TOKEN),
        "array",         token(ARRAY_TOKEN),
        "begin",         token(BEGIN_TOKEN),
        "case",          token(CASE_TOKEN),
        "const",         token(CONST_TOKEN),
        "div",           token(DIV_TOKEN),
        "do",            token(DO_TOKEN),
        "downto",        token(DOWNTO_TOKEN),
        "else",          token(ELSE_TOKEN),
```

```
        "end",          token(END_TOKEN),
        "external",     token(EXTERNAL_TOKEN),
        "extern",       token(EXTERNAL_TOKEN),
        "file",         token(FILE_TOKEN),
        "for",          token(FOR_TOKEN),
        "forward",      token(FORWARD_TOKEN),
        "function",     token(FUNCTION_TOKEN),
        "goto",         token(GOTO_TOKEN),
        "if",           token(IF_TOKEN),
        "in",           token(IN_TOKEN),
        "label",        token(LABEL_TOKEN),
        "mod",          token(MOD_TOKEN),
        "nil",          token(NIL_TOKEN),
        "not",          token(NOT_TOKEN),
        "of",           token(OF_TOKEN),
        "or",           token(OR_TOKEN),
        "packed",       token(PACKED_TOKEN),
        "procedure",    token(PROCEDURE_TOKEN),
        "program",      token(PROGRAM_TOKEN),
        "record",       token(RECORD_TOKEN),
        "repeat",       token(REPEAT_TOKEN),
        "set",          token(SET_TOKEN),
        "then",         token(THEN_TOKEN),
        "to",           token(TO_TOKEN),
        "type",         token(TYPE_TOKEN),
        "until",        token(UNTIL_TOKEN),
        "var",          token(VAR_TOKEN),
        "while",        token(WHILE_TOKEN),
        "with",         token(WITH_TOKEN),

        };


static int screen()
{       struct rwtable * low = rwtable,
                * high = END(rwtable),
                * mid;
        int c;
        int subscript;
        char character;
        /* convert yytext to lower case if necessary */

        subscript = 0;
        while (subscript < (YYLMAX+1))
        {
        character = yytext[subscript];
        if (character >= 'A' && character <= 'Z') /*it is in uppercase*/
        {
           character = ( character + 'a' - 'A' );
```

```
            loweryytext[subscript] = character;
        }
        loweryytext[subscript] = character;
        subscript = subscript + 1;

        }
        while (low <= high)
        {        mid = low + (high-low)/2;
                 if ((c = strcmp(mid->rw_name,loweryytext)) == 0)
                         return mid->rw_yylex;
                 else if (c < 0)
                          low = mid+1;
                 else
                          high = mid-1;
        }
        return token(IDENTIFIER_TOKEN);
}


yywrap()
{
  printf("\nEnd of file encountered line: %d\n",yylineno);
}
```

Appendix B

A Pascal Parser Specification

```
/*
 *       Yacc Grammar
 *       Backus Naur Form Specification for
 *       Computer programming language Pascal
 *
 *       Date    : 12/1/89
 *       Project : Static Analysis Tools for use in Software Maintenance
 *       (shift/reduce conflicts : one on ELSE_TOKEN)
 */

/*
 *       terminal symbols
 */
%token   IDENTIFIER_TOKEN
%token   CONSTANT_TOKEN
%token   END_OF_FILE_TOKEN
%token   QUOTE_TOKEN     /* ' */
%token   EXTERNAL_TOKEN
%token   EXTERN_TOKEN
%token   STRING_TOKEN
%token   REAL_TOKEN
%token   INTEGER_TOKEN
%token   NE_TOKEN        /* <> */
%token   LE_TOKEN        /* <= */
%token   GE_TOKEN        /* >= */
%token   AS_TOKEN        /* := */
%token   EL_TOKEN        /* .. */
```

```
%token    AND_TOKEN
%token    ARRAY_TOKEN
%token    BEGIN_TOKEN
%token    CASE_TOKEN
%token    CONST_TOKEN
%token    DIV_TOKEN
%token    DO_TOKEN
%token    DOWNTO_TOKEN
%token    ELSE_TOKEN
%token    END_TOKEN
%token    EXTERNAL_TOKEN
%token    FILE_TOKEN
%token    FOR_TOKEN
%token    FORWARD_TOKEN
%token    FUNCTION_TOKEN
%token    GOTO_TOKEN
%token    IF_TOKEN
%token    IN_TOKEN
%token    INCLUDE_TOKEN
%token    LABEL_TOKEN
%token    MOD_TOKEN
%token    NIL_TOKEN
%token    NOT_TOKEN
%token    OF_TOKEN
%token    OR_TOKEN
%token    PACKED_TOKEN
%token    PROCEDURE_TOKEN
%token    PROGRAM_TOKEN
%token    RECORD_TOKEN
%token    REPEAT_TOKEN
%token    SET_TOKEN
%token    THEN_TOKEN
%token    TO_TOKEN
%token    TYPE_TOKEN
%token    UNTIL_TOKEN
%token    VAR_TOKEN
%token    WHILE_TOKEN
%token    WITH_TOKEN

/*
 *        precedence table
 */

%left     '=' NE_TOKEN
%left     '<' '>' GE_TOKEN LE_TOKEN
%left     '+' '-'
%left     '*' '/'
%left     NOT_TOKEN
```

```
%start   program

%%       /* beginning of rules section */

program
         : program_heading                    {INITIALISE_CROSS_REFERENCER();}
           block '.' /* END_OF_FILE_TOKEN*/   {PRINT_INTERMEDIATE_LISTING();}
         | declarations
         ;
block
         : {ENTER_SCOPE();}
           declarations statement_part {FINISHED_WITH_SCOPE();}
         | statement_part
         ;
declarations
         : declaration_part
         | declarations declaration_part
         ;
declaration_part
         : label_declaration_part
         | constant_definition_part
         | type_definition_part
         | variable_declaration_part
         | procedure_and_function_declaration_part
         | includes
         ;

label_declaration_part
         : LABEL_TOKEN {LABEL_CLASS();}
           label_part ';'
         ;
label_part
         : INTEGER_TOKEN                 {CONNECT_TO_OBJECT(current_text);
                                          NAMESPACE(source_file_name);
                                          CLASS();
                                          DISCONNECT_FROM_OBJECT();}

         | label_part ',' INTEGER_TOKEN {CONNECT_TO_OBJECT(current_text);
                                          NAMESPACE(source_file_name);
                                          CLASS();
                                          DISCONNECT_FROM_OBJECT();}

         ;
constant_definition_part
         : CONST_TOKEN                  {CONSTANT_CLASS();}
           constant_part ';'
         ;
constant_part
         : IDENTIFIER_TOKEN            {CONNECT_TO_OBJECT(current_text);
```

```
                                        NAMESPACE(source_file_name);}
          '=' constant                  {CONSTANT_VALUE(text_buffer;}
        | constant_part ';'
         IDENTIFIER_TOKEN               {CONNECT_TO_OBJECT(current_text);
                                        NAMESPACE(source_file);
                                        DISCONNECT_FROM_OBJECT();}
          '='   {ADD_TO_TEXT_BUFFER(current_text);}
          constant
        ;
constant
        : sign INTEGER_TOKEN      {ADD_TO_TEXT_BUFFER(current_text);}
        | INTEGER_TOKEN           {ADD_TO_TEXT_BUFFER(current_text);}
        | sign IDENTIFIER_TOKEN   {ADD_TO_TEXT_BUFFER(current_text);}
        | IDENTIFIER_TOKEN        {ADD_TO_TEXT_BUFFER(current_text);}
        | STRING_TOKEN            {ADD_TO_TEXT_BUFFER(current_text);}
        | '+' REAL_TOKEN          {ADD_TO_TEXT_BUFFER(current_text);}
        | '-' REAL_TOKEN          {ADD_TO_TEXT_BUFFER(current_text);}
        | REAL_TOKEN              {ADD_TO_TEXT_BUFFER(current_text);}
        ;
sign    : '+' {ADD_TO_TEXT_BUFFER(current_text);}
        | '-' {ADD_TO_TEXT_BUFFER(current_text);}
        ;
type_definition_part
        : TYPE_TOKEN, {TYPE_CLASS();}
          type_part ';'
        ;
type_part
        : IDENTIFIER_TOKEN       {CONNECT_TO_OBJECT(Identifier);
                                 NAMESPACE(source_file_name);
                                 CLASS();
                                 ADD_TO_TEXT_BUFFER(current_text);}
          '='                    {ADD_TO_TEXT_BUFFER(current_text);}
          type
        | type_part
          ';'                    {ADD_TO_TEXT_BUFFER(current_text);}
                IDENTIFIER_TOKEN  {CONNECT_TO_OBJECT(current_text);
                                  NAMESPACE(source_file_name);
                                  CLASS();
                                  DECLARED_ON(line_number);}
          '='                     {ADD_TO_TEXT_BUFFER(current_text);}
          type                    {TYPE_VALUE();}
        ;
type
        : simple_type
        | structured_type
        | pointer_type
        ;
simple_type
        : scalar_type
```

```
          | subrange_type
          | IDENTIFIER_TOKEN      {CONNECT_TO_OBJECT(Identifier);
                                  USED_ON(line_number);}
          ;
scalar_type
          : '('                   {ADD_TO_TEXT_BUFFER(Current_text);}
             scalar_part ')'      {ADD_TO_TEXT_BUFFER(Current_text);}
          ;
scalar_part
          : IDENTIFIER_TOKEN      {CONNECT_TO_OBJECT(Identifier);
                                  USED_ON(line_number);}
          | scalar_part ','       {ADD_TO_TEXT_BUFFER(Current_text);}
            IDENTIFIER_TOKEN      {CONNECT_TO_OBJECT(Identifier);
                                  USED_ON(line_number);}
          ;
subrange_type
          : constant EL_TOKEN     {ADD_TO_TEXT_BUFFER(Current_text);}
            constant
          ;
structured_type
          : packed array_type
          | packed record_type
          | packed set_type
          | packed file_type
          | array_type
          | record_type
          | set_type
          | file_type
          ;
packed
          : PACKED_TOKEN      {ADD_TO_TEXT_BUFFER(Current_text);}
          ;
array_type
          : ARRAY_TOKEN       {ADD_TO_TEXT_BUFFER(Current_text);}
           '['                {ADD_TO_TEXT_BUFFER(Current_text);}
            simple_type_part
           ']'                {ADD_TO_TEXT_BUFFER(Current_text);}
           OF_TOKEN           {ADD_TO_TEXT_BUFFER(Current_text);}
           type
          ;
simple_type_part
          : simple_type
          | simple_type_part ',' {ADD_TO_TEXT_BUFFER(Current_text);}
            simple_type
          ;
record_type
          : RECORD_TOKEN        {ADD_TO_TEXT_BUFFER(Current_text);}
            field_list end_part
          ;
```

```
field_list
        : fixed_part
        | fixed_part ';'      {ADD_TO_TEXT_BUFFER(Current_text);}
          variant_part
        | variant_part
        ;
fixed_part
        : field_identifier ':'  {ADD_TO_TEXT_BUFFER(Current_text);}
          type
        | fixed_part ';'        {ADD_TO_TEXT_BUFFER(Current_text);}
          field_identifier ':'  {ADD_TO_TEXT_BUFFER(Current_text);}
          type
        ;
field_identifier
        : IDENTIFIER_TOKEN      {ADD_TO_TEXT_BUFFER(Current_text);
                                 CONNECT_TO_OBJECT(Identifier);
                                 NAMESPACE(source_file_name);
                                 CLASS();}


        | field_identifier  ',' {ADD_TO_TEXT_BUFFER(Current_text);}
          IDENTIFIER_TOKEN      {ADD_TO_TEXT_BUFFER(Current_text);
                                 CONNECT_TO_OBJECT(Identifier);
                                 NAMESPACE(source_file_name);
                                 CLASS();}

        ;
variant_part
        : CASE_TOKEN            {ADD_TO_TEXT_BUFFER(Identifier);}
          case_variant_selector
          OF_TOKEN              {ADD_TO_TEXT_BUFFER(Identifier);}
          variant_component
        ;
case_variant_selector
        : IDENTIFIER_TOKEN   {CONNECT_TO_OBJECT(Identifier);
                              NAMESPACE(source_file_name);
                              CLASS();
                              DISCONNECT_FROM_OBJECT();}

          ':'                {ADD_TO_TEXT_BUFFFER(Identifier);}
          IDENTIFIER_TOKEN   {CONNECT_TO_OBJECT(Identifier);
                              NAMESPACE(source_file_name);
                              CLASS();
                              DISCONNECT_FROM_OBJECT();}

        | IDENTIFIER_TOKEN   {CONNECT_TO_OBJECT(Identifier);
                              NAMESPACE(source_file_name);
                              CLASS();
                              DISCONNECT_FROM_OBJECT();}
```

```
        ;
variant_component
        : variant
        |
        | variant  ';'        {ADD_TO_TEXT_BUFFER(Identifier);}
          variant_component
        ;
variant
        : case_label_list ':' {ADD_TO_TEXT_BUFFER(Identifier);}
         '('                  {ADD_TO_TEXT_BUFFER(Identifier);}
         field_list ')'       {ADD_TO_TEXT_BUFFER(Identifier);}
        ;
case_label_list
        : constant
        | case_label_list ',' {ADD_TO_TEXT_BUFFER(Identifier);}
          constant
        ;
set_type
        : SET_TOKEN           {ADD_TO_TEXT_BUFFER(Identifier);}
          OF_TOKEN              {ADD_TO_TEXT_BUFFER(Identifier);}
          simple_type
        ;
file_type
        : FILE_TOKEN            {ADD_TO_TEXT_BUFFER(Identifier);}
          OF_TOKEN              {ADD_TO_TEXT_BUFFER(Identifier);}
          type
        ;
pointer_type
        : '^' IDENTIFIER_TOKEN  {ADD_TO_TEXT_BUFFER(Identifier);
                                 CONNECT_TO_OBJECT(Identifier);
                                 NAMESPACE(source_file_name);
                                 DECLARED_ON(line_number);
                                 DISCONNECT_FROM_OBJECT();}

        ;
variable_declaration_part
        : VAR_TOKEN             {VAR_CLASS();}
          variable_identifier_part ';'
        ;
variable_identifier_part
        : identifier_list ':'
          type                {TYPE_CLASS();}
        | variable_identifier_part
          ';' identifier_list ':'
          type                {TYPE_CLASS();}
        ;
identifier_list
        : IDENTIFIER_TOKEN     {CONNECT_TO_OBJECT(Identifier);
```

```
                                NAMESPACE(source_file_name);
                                DECLARED_ON(line_number);
                                DISCONNECT_FROM_OBJECT();}

          | identifier_list  ','

            IDENTIFIER_TOKEN     {CONNECT_TO_OBJECT(Identifier);
                                 NAMESPACE(source_file_name);
                                 DECLARED_ON(line_number);
                                 DISCONNECT_FROM_OBJECT();}

          ;
parameter_identifier_list
          : IDENTIFIER_TOKEN    {CONNECT_TO_OBJECT(Identifier);
                                NAMESPACE(source_file_name);
                                PARAMETER_ON(line_number);
                                DISCONNECT_FROM_OBJECT();}
          | parameter_identifier_list ','
            IDENTIFIER_TOKEN    {CONNECT_TO_OBJECT(Identifier);
                                NAMESPACE(source_file_name);
                                PARAMETER_ON(line_number);
                                DISCONNECT_FROM_OBJECT();}
          ;
procedure_and_function_declaration_part
          : procedure_declaration
          | function_declaration
          ;
procedure_declaration
          : procedure_heading {DISCONNECT_FROM_OBJECT();} ';' block ';'
          | procedure_heading {DISCONNECT_FROM_OBJECT();} ';' external
          ;
function_declaration
          : function_heading {DISCONNECT_FROM_OBJECT();} ';' block ';'
          | function_heading ';' external {DISCONNECT_FROM_OBJECT();}
          ;
external
          : EXTERNAL_TOKEN {EXTERNAL_ON(linenumber);} ';'
          | EXTERNAL_TOKEN {EXTERNAL_ON(linenumber);}
          | FORWARD_TOKEN {FORWARD_ON(linenumber);}
          | FORWARD_TOKEN  {FORWARD_ON(linenumber);} ';'
          ;
procedure_heading
          : PROCEDURE_TOKEN IDENTIFIER_TOKEN  {CONNECT_TO_OBJECT(Identifier);
                                              PROCEDURE_CLASS():
                                              NAMESPACE(source_file_name);
                                              DECLARED_ON(linenumber);
                                              }

          '(' parameter_section ')'
```

```
        | PROCEDURE_TOKEN IDENTIFIER_TOKEN {CONNECT_TO_OBJECT(Identifier);
                                            PROCEDURE_CLASS():
                                            NAMESPACE(source_file_name);
                                            DECLARED_ON(linenumber);
                                            }

        ;
parameter_section
        : formal_parameter_section
        | parameter_section ';' formal_parameter_section
        ;
function_heading
        : FUNCTION_TOKEN IDENTIFIER_TOKEN  {CONNECT_TO_OBJECT(Identifier);
                                           FUNCTION_CLASS():
                                           NAMESPACE(source_file_name);
                                           DECLARED_ON(linenumber);
                                           }

          '(' parameter_section ')'
          function_result
        | FUNCTION_TOKEN IDENTIFIER_TOKEN {CONNECT_TO_OBJECT(Identifier);
                                           FUNCTION_CLASS();
                                           NAMESPACE(source_file_name);
                                           DECLARED_ON(linenumber);
                                           }

          function_result
        | FUNCTION_TOKEN IDENTIFIER_TOKEN  {CONNECT_TO_OBJECT(Identifier);
                                          FUNCTION_CLASS():
                                          NAMESPACE(source_file_name);
                                          DECLARED_ON(linenumber);
                                          DISCONNECT_FROM_OBJECT();}

        ;
function_result
        : ':' IDENTIFIER_TOKEN          {CONNECT_TO_OBJECT(Identifier);
                                        NAMESPACE(source_file_name);
                                        FUNCTION_RETURN_TYPE(linenumber);
                                        DISCONNECT_FROM_OBJECT();}

        ;
formal_parameter_section
        : identifier_list ':'
          IDENTIFIER_TOKEN              {CONNECT_TO_OBJECT(Identifier);
                                        NAMESPACE(source_file_name);
                                        PASCAL_TYPE(linenumber);
                                        DISCONNECT_FROM_OBJECT();}
        | VAR_TOKEN identifier_list ':'
          IDENTIFIER_TOKEN              {CONNECT_TO_OBJECT(Identifier);
```

```
                                        NAMESPACE(source_file_name);
                                        PASCAL_TYPE(linenumber);
                                        DISCONNECT_FROM_OBJECT();}

          | procedure_heading
          | function_heading
          ;
result_type
          : IDENTIFIER_TOKEN
          ;
statement_part
          : compound_statement
          ;
compound_statement
          : BEGIN_TOKEN statement_sequence end_part
          ;
end_part
          : END_TOKEN
          | ';' END_TOKEN
          ;
statement_sequence
          : statement
          | statement_sequence ';'  statement
          ;
statement
          : simple_statement
          | structured_statement
          | label_part ':' simple_statement
          | label_part ':' structured_statement
          | labels
          | /* null */
          ;
labels
          : INTEGER_TOKEN           {LABEL_CLASS();
                                    CONNECT_TO_OBJECT(Identifier)
                                    NAMESPACE(source_file_name);
                                    USED_ON(line_number);
                                    DISCONNECT_FROM_OBJECT();}
            ':'
             | labels INTEGER_TOKEN {LABEL_CLASS();
                                     CONNECT_TO_OBJECT(Identifier)
                                     NAMESPACE(source_file_name);
                                     USED_ON(line_number);
                                     DISCONNECT_FROM_OBJECT();}

             ':'
          ;
simple_statement
          : assignment_statement
```

```
          | procedure_statement
          | goto_statement
          ;
structured_statement
          : compound_statement
          | conditional_statement
          | repetitive_statement
          | with_statement
          ;
assignment_statement
          : variable AS_TOKEN expression
          ;
variable
          : variable_head
          | variable '.' variable_head
          ;
variable_head
          : IDENTIFIER_TOKEN        {CONNECT_TO_OBJECT(Identifier);
                                    NAMESPACE(source_file_name);
                                    SET_ON(line_number);
                                    DISCOUNT_FROM_OBJECT();}
            '[' subscripts ']'
          | IDENTIFIER_TOKEN \* only record id on bottom level*\
            '^'
          | IDENTIFIER_TOKEN
            '@'
          | IDENTIFIER_TOKEN        {CONNECT_TO_OBJECT(Identifier);
                                    NAMESPACE(source_file_name);
                                    SET_ON(line_number);
                                    DISCOUNT_FROM_OBJECT();}
          ;
subscripts
          : expression
          | subscripts ',' expression
          ;
expression
          : simple_expression
          | simple_expression '=' simple_expression
          | simple_expression NE_TOKEN simple_expression
          | simple_expression '<' simple_expression
          | simple_expression LE_TOKEN simple_expression
          | simple_expression GE_TOKEN simple_expression
          | simple_expression '>' simple_expression
          | simple_expression IN_TOKEN simple_expression
          ;
simple_expression
          : simple_expression_term
          | '+' simple_expression_term
          | '-' simple_expression_term
```

```
          ;
 simple_expression_term
          : term
          | simple_expression_term '+' term
          | simple_expression_term '-' term
          | simple_expression_term  OR_TOKEN term
          ;
 term
          : factor
          | term '*' factor
          | term '/' factor
          | term DIV_TOKEN factor
          | term MOD_TOKEN factor
          | term AND_TOKEN factor
          ;
 factor
          : variable_or_factor
          | unsigned_constant
          | '(' expression ')' field_width
          | '(' expression ')'
          | set
          | factor variable_or_factor
          | NOT_TOKEN factor
          ;
 variable_or_factor
          : IDENTIFIER_TOKEN                  {CONNECT_TO_OBJECT(Identifier);
                                              NAMESPACE(source_file_name);
                                              USED_ON(linenumber);
                                              DISCONNECT_FROM_OBJECT();}
            '(' actual_parameter_list ')' field_width
          | IDENTIFIER_TOKEN                  {CONNECT_TO_OBJECT(Identifier);
                                               NAMESPACE(source_file_name);
                                               USED_ON(linenumber);
                                               DISCONNECT_FROM_OBJECT();}
 '(' actual_parameter_list ')'
          | variable
          | variable field_width
          ;
 field_width
          : ':' field_width_parameter
          | ':' variable
          | ':' variable ':' variable
          | ':' variable ':' field_width_parameter
          | ':' field_width_parameter ':' field_width_parameter
          | ':' field_width_parameter ':' variable
          ;
 field_width_parameter
          : sign INTEGER_TOKEN
          | INTEGER_TOKEN
```

```
        ;
unsigned_constant
        : NIL_TOKEN
        | unsigned_number
        | unsigned_number field_width
        | STRING_TOKEN
        | STRING_TOKEN field_width
        ;
set
        : '[' ']'
        | '[' set_part ']'
        ;
set_part
        : element
        | set_part ','  element
        ;
element
        : expression
        | expression  EL_TOKEN  expression
        ;
procedure_statement
        : IDENTIFIER_TOKEN                  {CONNECT_TO_OBJECT(Identifier);
                                            CALLED_ON(linenumber);
                                            DISCONNECT_FROM_OBJECT(); }
          '(' actual_parameter_list ')'
        | IDENTIFIER_TOKEN                  {CONNECT_TO_OBJECT(Identifier);
                                            CALLED_ON(linenumber);
                                            DISCONNECT_FROM_OBJECT(); }

        ;
actual_parameter_list
        : actual_parameter
        | actual_parameter_list ',' actual_parameter
        ;
actual_parameter
        : expression
        ;
goto_statement
        : GOTO_TOKEN label_part
        ;
conditional_statement
        : if_statement
        | case_statement
        ;
if_statement
        : IF_TOKEN expression THEN_TOKEN statement
        | IF_TOKEN expression THEN_TOKEN statement ELSE_TOKEN statement
        ;
case_statement
```

```
        : CASE_TOKEN  expression OF_TOKEN  case_list_element_part  END_TOKEN

        ;
case_list_element_part
        : case_list_element
        | case_list_element_part  ';' case_list_element
        ;
case_list_element
        : case_label_list ':' statement
        ;
repetitive_statement
        : WHILE_TOKEN expression DO_TOKEN statement
        | REPEAT_TOKEN statement_sequence UNTIL_TOKEN expression
        | FOR_TOKEN  IDENTIFIER_TOKEN       {CONNECT_TO_OBJECT(Identifier);
                                             NAMESPACE(name_of_source_file);
                                             IN_FOR_LOOP_ON(line_number);
                                             DISCONNECT_FROM_OBJECT();}
          AS_TOKEN for_list DO_TOKEN statement
        ;
for_list
        : expression TO_TOKEN expression
        | expression DOWNTO_TOKEN  expression
        ;
with_statement
        : WITH_TOKEN                         {IN_WITH_ON(line_number);}
          with_variable_list DO_TOKEN statement
        ;
with_variable_list
        : variable
        | with_variable_list ',' variable
        ;
unsigned_number
        : INTEGER_TOKEN
        | REAL_TOKEN
        ;
program_heading
        : PROGRAM_TOKEN program_name file_assignments ';'
        ;

program_name
        : IDENTIFIER_TOKEN
        ;

file_assignments
        : '(' file_identifiers ')'
        |
        ;
file_identifiers
        : IDENTIFIER_TOKEN
```

```
        | file_identifiers ',' IDENTIFIER_TOKEN
        ;
includes
        : INCLUDE_TOKEN STRING_TOKEN
        | INCLUDE_TOKEN STRING_TOKEN ';'
        ;
```

Appendix C

Pascal Cross Referencer Specification

```
CROSS_REFERENCER_SPECIFICATION;   /* for pascalpar.y  version 1.0 */

    OBJECT_CLASSES;

        <parameter_class>
               selected_by_routine parameter_class();
               intermediate_rep_name ".0parameter";
        <label_class>
               selected_by_routine label_class();
               intermediate_rep_name ".4label";
        <constant_class>
               selected_by_routine constant_class();
               intermediate_rep_name ".5constant";
        <type_class>
               selected_by_routine type_class();
               intermediate_rep_name ".6type";
        <var_class>
               selected_by_routine var_class();
               intermediate_rep_name ".7variable";
        <function_class>
               selected_by_routine function_class();
               intermediate_rep_name ".8function";
               when_same   forward_def;
        <procedure_class>
               selected_by_routine procedure_class();
               intermediate_rep_name ".9procedure";
```

```
        when_same   forward_def;
    end;


DATA_ITEMS_PASSED_FROM_PARSER;

    <char *>  character_string;
    <char *>  source_file_name;
    <int>     line_number;
    end;



SEMANTIC_ROUTINES;

    [1]  object(character_string);
    [2]  namespace(source_file_name);
    [3]  var_as_parameter(line_number);
    [4]  class();
    [5]  pascal_type(character_string);
    [6]  function_return_type(character);
    [7]  external(line_number);
    [8]  forward(line_number);
    [9]  const_value(character_string);
    [10] non_local(non_local_object);
    [11] declared(line_number);
    [12] set(line_number);
    [13] used(line_number);
    [14] inFORloop(line_number);
    [15] parameter_on(line_number);
    [16] inWITH(line_number);
    [17] called(line_number);
    [18] undefined(line_number);
    [19] ingoto(line_number);
    end;


SEMANTIC_ROUTINE_USAGE;

    <parameter_class> uses_semantic_routines  [1,2,4,12,15,16]
    <label_class>     uses_semantic_routines  [1,2,4,11,18,19]
    <const_class>     uses_semantic_routines  [1,2,4,9,11,13]
    <type_class>      uses_semantic_routines  [1,2,4,5,11,13]
    <var_class>       uses_semantic_routines  [1,2,3,4,5,11,12,14,16]
    <function_class>  uses_semantic_routines  [1,2,3,4,6,7,8,10,15,17]
    <procedure_class> uses_semantic_routines  [1,2,3,4,7,8,10,17]
    end;
```

```
TEXT_BUFFER_USAGE;

    /* functions which get data items via text buffer*/
    const_value();
    pascal_type();
    end;


CROSS_REFERENCER_OBJECT_STRUCTURES;

    <parameter_class> slot_structure_is
           (object,
           namespace,
           class,
           pascal_type,
           parameter_on,
           inwith)
    <label_class>  slot_structure_is
           (object,
           namespace,
           class,
           declared,
           undefined,
           ingoto)
    <const_class>  slot_structure_is
           (object,
           namespace,
           class,
           const_value,
           declared,
           used)
   <type_class>   slot_structure_is
           (object,
           namespace,
           class,
           pascal_type,
           declared,
           used)
   <var_class>    slot_structure_is
           (object,
           namespace,
           class,
           pascal_type,
           declared,
           used,
           var_as_parameter,
           inwith,
           infor)
   <procedure_class> slot_structure_is
```

```
        (object,
        namespace,
        class,
        parameter_on,
        external,
        forward,
        non_local,
        called)
  <function_class>  slot_structure_is
        (object,
        namespace,
        class,
        parameter_on,
        external,
        forward,
        non_local,
        called,
        function_return_type)
  end;


OBJECT_SLOT_DESCRIPTIONS;

  object  :
        type_id         is   character_string
        type            is   char
        slot_value      is   single
        meta_slot_value is   .object
        end object

  namespace:
        type_id         is   character_string
        type            is   char
        slot_value      is   single
        meta_slot_value is   =0namespace
        end namespace

  var_as_parameter :
        type_id         is   line_number
        type            is   int
        slot_value      is   multi
        meta_slot_value is  .0parameter
        end var_as_parameter

  class :
        type_id         is   character_string
        type            is   char
        slot_value      is   single
        meta_slot_value is   =1class
```

```
        end class

pascal_type  :
        type_id         is   character_string
        type            is   char
        slot_value      is   single
        meta_slot_value is   =2type
        end pascal_type

function_return_type :
        type_id         is   character_string
        type            is   char
        slot_value      is   single
        meta_slot_value is   +000type
        end function_return_type

external  :
        type_id        is    line_number
        type            is   int
        slot_value      is   single
        meta_slot_value is   +001external
        end external

forward :
        type_id         is   line_number
        type            is   int
        slot_value      is   single
        meta_slot_value is   +002forward
        end forward
const_value :
        type_id         is   character_string
        type            is   char
        slot_value      is   single
        meta_slot_value is   +01value
        end const_value

non_local :
        type_id         is   character_string
        type            is   char
        slot_value      is   single
        meta_slot_value is   +02non_local
        end non_local

declared :
        type_id         is   line_number
        type            is   int
        slot_value      is   single
        meta_slot_value is   +03declared
        end declared
```

```
 set :
        type_id          is   line_number
        type             is   int
        slot_value       is   multi
        meta_slot_value is   +04set
        end set

 used :
        type_id          is   line_number
        type             is   int
        slot_value       is   multi
        meta_slot_value is  +05used
        end used

infor :
        type_id          is   line_number
        type             is   int
        slot_value       is   multi
        meta_slot_value is   +06infor
        end infor

parameter_on :
        type_id          is   line_number
        type             is   int
        slot_value       is   multi
        meta_slot_value is  +07parameter
        end parameter_on

inwith :
        type_id          is  line_number
        type             is  int
        slot_value       is  multi
        meta_slot_value is  +08inwith
        end inwith

called :
        type_id          is  line_number
        type             is  int
        slot_value       is  multi
        meta_slot_value is  +09called
        end called

undefined  :
        type_id          is  line_number
        type             is  int
        slot_value       is  multi
        meta_slot_value is  +10undefined
        end undefined
```

```
    ingoto :
            type_id         is  line_number
            type            is  int
            slot_value      is  multi
            meta_slot_value is  +11ingoto
            end ingoto

    end;


SOURCE_CODE_IDENTIFIERS_TO_BE_IGNORED;

    odd;
    abs;
    sqr;
    trunc;
    round;
    sqrt;
    arctan;
    cos;
    sin;
    exp;
    ln;
    succ;
    pred;
    ord;
    chr;
    eof;
    reset;
    get;
    read;
    readln;
    rewrite;
    put;
    write;
    writeln;

    new;
    dispose;
    pack;
    unpack;
    end;

LISTING_SPECIFICATION;  /* Object packets to be included in listing */

    TERSE_SPEC;
      object;
      namespace;
```

```
        class;
        declared;
        set;
        used;
      end;

      INTERMEDIATE_SPEC;
        object;
        namespace;
        class;
        declared;
        set;
        used;
        undefined;
        inwith;
        infor;
        ingoto;
      end;

   end; /* listing specification */

END_OF_CROSS_REFERENCER_SPECIFICATION   /* pascalpar.y version 1.0 */
```

Appendix D

Lexical Analyser Specification for the Cross Referencer Notation

```
%{
/*
 *   Lex specification
 *
 *
 *    for cross referencer specification analyser (version 1.0) .
 *    Project : Software Maintenance: Generating Front Ends for
 *              Cross Reference Tools
 *    Date    :  2/5/1989
 */

#define END(v)  (v-1 + sizeof v /sizeof v[0])

int linenumber  =1;
char loweryytext[YYLMAX];
static int screen();
%}
letter                          [a-zA-Z]
digit                           [0-9]
letter_or_digit                 [a-zA-Z_0-9]
white_space                     [ \t\n]
blank                           [ \t]
return                          [\n]
double_quote                    ["]
```

```
%%
{letter}{letter_or_digit}*                    {return screen();}
"."({letter_or_digit}|"-"|"_")+              {return token(META_IDENTIFIER_TOKEN);}
"="({letter_or_digit}|"-"|"_")+              {return token(META_IDENTIFIER_TOKEN);}
"+"({letter_or_digit}|"-"|"_")+              {return token(META_IDENTIFIER_TOKEN);}
[0-9]*                                        {return token(INTEGER_TOKEN);}
{letter}({letter_or_digit}|"-"|"_")*          {return token(IDENTIFIER_TOKEN) ;}
{blank}                                       {}
{white_space}                                 { linenumber  = linenumber + 1; }
"'"                                           { return token(QUOTE_TOKEN); }
.                                             { return token(yytext[0]); }
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"      {}


%%


/*
 *       reserved word screener
 */

static struct rwtable {          /* reserved word table */
        char * rw_name;          /* representation */
        int rw_yylex;            /* yylex() value */
        } rwtable[] = {          /* sorted */
        "char",                              token(CHAR_TOKEN),
        "character",                         token(CHARACTER_TOKEN),
        "characters",                        token(CHARACTER_TOKEN),
        "cross_referencer_object_structures",
                                token(CROSS_REFERENCER_OBJECT_STRUCTURES_TOKEN),
        "cross_referencer_specification",    token(CROSS_REFERENCER_START_TOKEN),
        "data_items_passed_from_parser",     token(DATA_ITEMS_PASSED_FROM_PARSER_TOKEN),
        "defined_in_child_scope",            token(DEFINED_IN_CHILD_SCOPE_TOKEN),
        "defined_in_outer_scope",            token(DEFINED_IN_OUTER_SCOPE_TOKEN),
        "defined_in_parent_scope",           token(DEFINED_IN_PARENT_SCOPE_TOKEN),
        "edit_is",                           token(EDIT_IS_TOKEN),
        "end",                               token(END_TOKEN),
        "end_of_cross_referencer_specification",
                                token(END_OF_CROSS_REFERENCER_SPECIFICATION_TOKEN),
        "forward_def",                       token(INCOMPLETE_DEFINITION_TOKEN),
        "int",                               token(INT_TOKEN),
        "intermediate_rep_name",             token(INTERMEDIATE_REP_NAME_TOKEN),
        "intermediate_spec",                 token(INTERMEDIATE_SPEC_TOKEN),
        "is",                                token(IS_TOKEN),
        "listing_specification",             token(LISTING_SPECIFICATION_TOKEN),
        "lowercase",                         token(LOWERCASE_TOKEN),
        "meta_slot_value",                   token(META_SLOT_VALUE_TOKEN),
        "multi",                             token(MULTI_TOKEN),
```

```
        "no_truncation",                    token(NO_TRUNCATION_TOKEN),
        "object_class",                     token(OBJECT_CLASS_TOKEN),
        "object_classes",                   token(OBJECT_CLASSES_TOKEN),
        "object_look_up_edit_definition",   token(OBJECT_LOOK_UP_EDIT_DEFINITION),
        "object_slot_descriptions",         token(OBJECT_SLOT_DESCRIPTIONS_TOKEN),
        "redefine",                         token(REDEFINE_TOKEN),
        "same_as_source",                   token(SAME_AS_SOURCE_TOKEN),
        "scope_classes",                    token(SCOPE_CLASSES_TOKEN),
        "selected_by_routine",              token(SELECTED_BY_ROUTINE_TOKEN),
        "semantic_routine_usage",           token(SEMANTIC_ROUTINE_USAGE_TOKEN),
        "semantic_routines",                token(SEMANTIC_ROUTINES_TOKEN),
        "single",                           token(SINGLE_TOKEN),
        "slot_structure_is",                token(SLOT_STRUCTURE_IS_TOKEN),
        "slot_value",                       token(SLOT_VALUE_TOKEN),
        "source_code_identifiers_to_be_ignored",
                            token(SOURCE_CODE_IDENTIFIERS_TO_BE_IGNORED_TOKEN),
        "terse_spec",                       token(TERSE_SPEC_TOKEN),
        "text_buffer_usage",                token(TEXT_BUFFER_USAGE_TOKEN),
        "truncation_after",                 token(TRUNCATION_AFTER_VALUE_TOKEN),
        "type",                             token(TYPE_TOKEN),
        "type_id",                          token(TYPE_ID_TOKEN),
        "uppercase",                        token(UPPERCASE_TOKEN),
        "uses_semantic_routines",           token(USES_SEMANTIC_ROUTINES_TOKEN),
        "visible_in_child_scope",           token(VISIBLE_IN_CHILD_SCOPE_TOKEN),
        "visible_in_outer_scope",           token(VISIBLE_IN_OUTER_SCOPE_TOKEN),
        "visible_in_parent_scope",          token(VISIBLE_IN_PARENT_SCOPE_TOKEN),
        "visible_in_program",               token(VISIBLE_IN_PROGRAM_TOKEN),
        "when_nested",                      token(WHEN_NESTED_TOKEN),
        "when_same",                        token(WHEN_SAME_TOKEN),

         };

static int screen()
{       struct rwtable * low = rwtable,
                * high = END(rwtable),
                * mid;
        int c;
        int subscript;
        char character;
        /* convert yytext to lower case if necessary */

        subscript = 0;
        while (subscript < (YYLMAX+1))
        {
        character = yytext[subscript];
        if (character >= 'A' && character <= 'Z') /*it is in uppercase*/
        {
           character = ( character + 'a' - 'A' );
           loweryytext[subscript] = character;
```

```
        }
        loweryytext[subscript] = character;
        subscript = subscript + 1;
        }
        while (low <= high)
        {       mid = low + (high-low)/2;
                if ((c = strcmp(mid->rw_name,loweryytext)) == 0)
                        return mid->rw_yylex;
                else if (c < 0)
                         low = mid+1;
                else
                         high = mid-1;
        }
        return token(IDENTIFIER_TOKEN);
}


yywrap()
{
  printf("          End of file encountered line: %d\n",yylineno);

}
```

Appendix E

Grammar of the Cross Reference Specification

```
/*  Yacc Grammar for
 *
 *
 *
 *       Cross Referencer Specification Analyser (Version 1.0)
 *       Project : Static Analysis Tools for Software Maintenance
 *       Date    : 2/5/1989
 *
 */

/*
 *       This is the front end of the cross referencer generator
 *       It checks  the  specification is  syntactically correct
 *       and removes the  noise  words before creating  the xref
 *       product description files.
 *
 *
 *       The  xref product description files are the intermediate
 *       representation  of the cross  referencer to be generated.
 *       They are the input to xref product generator.
 *       The files are :
 *
 *              object_classes          class_selection
 *              intermediate_names      data_types
```

```
  *                type_identifiers        semantic_routines
  *                routine_usage           text_buffer_usage
  *                object_structure        slot_types
  *                terse_spec              intermediate_spec
  *                reserved_identifiers
  */

%{
extern char yytext[]; /* make lex.yy.c current token visible */
%}
 /*
  *       terminal symbols
  */
%token    META_IDENTIFIER_TOKEN
%token    INTEGER_TOKEN
%token    IDENTIFIER_TOKEN
%token    CHAR_TOKEN
%token    CHARACTER_TOKEN
%token    CROSS_REFERENCER_OBJECT_STRUCTURES_TOKEN
%token    CROSS_REFERENCER_START_TOKEN
%token    DATA_ITEMS_PASSED_FROM_PARSER_TOKEN
%token    DEFINED_IN_CHILD_SCOPE_TOKEN
%token    DEFINED_IN_OUTER_SCOPE_TOKEN
%token    DEFINED_IN_PARENT_SCOPE_TOKEN
%token    EDIT_IS_TOKEN
%token    END_TOKEN
%token    END_OF_CROSS_REFERENCER_SPECIFICATION_TOKEN
%token    INCOMPLETE_DEFINITION_TOKEN
%token    INT_TOKEN
%token    INTERMEDIATE_REP_NAME_TOKEN
%token    INTERMEDIATE_SPEC_TOKEN
%token    IS_TOKEN
%token    LISTING_SPECIFICATION_TOKEN
%token    LOWERCASE_TOKEN
%token    META_SLOT_VALUE_TOKEN
%token    MULTI_TOKEN
%token    NO_TRUNCATION_TOKEN
%token    OBJECT_CLASS_TOKEN
%token    OBJECT_CLASSES_TOKEN
%token    OBJECT_LOOK_UP_EDIT_DEFINITION
%token    OBJECT_SLOT_DESCRIPTIONS_TOKEN
%token    QUOTE_TOKEN
%token    REDEFINE_TOKEN
%token    SAME_AS_SOURCE_TOKEN
%token    SCOPE_CLASSES_TOKEN
%token    SELECTED_BY_ROUTINE_TOKEN
%token    SEMANTIC_ROUTINES_TOKEN
%token    SEMANTIC_ROUTINE_USAGE_TOKEN
%token    SINGLE_TOKEN
```

```
%token    SLOT_STRUCTURE_IS_TOKEN
%token    SLOT_VALUE_TOKEN
%token    SOURCE_CODE_IDENTIFIERS_TO_BE_IGNORED_TOKEN
%token    TERSE_SPEC_TOKEN
%token    TEXT_BUFFER_USAGE_TOKEN
%token    TRUNCATION_AFTER_VALUE_TOKEN
%token    TYPE_TOKEN
%token    TYPE_ID_TOKEN
%token    UPPERCASE_TOKEN
%token    USES_SEMANTIC_ROUTINES_TOKEN
%token    VISIBLE_IN_CHILD_SCOPE_TOKEN
%token    VISIBLE_IN_OUTER_SCOPE_TOKEN
%token    VISIBLE_IN_PARENT_SCOPE_TOKEN
%token    VISIBLE_IN_PROGRAM_TOKEN
%token    WHEN_NESTED_TOKEN
%token    WHEN_SAME_TOKEN


%start    cross_referencer_specification_grammar


%%        /* beginning of rules section */


cross_referencer_specification_grammar
          : spec_header  { init();}
            body         { close_product_files();}
            end_of_spec_token
          ;
body
          : object_classes
            data_items_passed_from_parser
            semantic_routines
            semantic_routine_usage
            text_buffer_usage
            object_structures
            object_slot_descriptions
            source_identifiers_to_be_ignored
            xref_listing_specification
          ;
spec_header
          : CROSS_REFERENCER_START_TOKEN ';'
          ;
object_classes
          : OBJECT_CLASSES_TOKEN ';' object_class_descriptions END_TOKEN ';'
          ;
object_class_descriptions
          : object_description
          | object_class_descriptions  object_description
```

```
          ;
object_description
          : '<' IDENTIFIER_TOKEN  {store_object_class(yytext);}
            '>' SELECTED_BY_ROUTINE_TOKEN
            IDENTIFIER_TOKEN {store_object_class_selection(yytext);}
            '(' ')' ';'
            INTERMEDIATE_REP_NAME_TOKEN '"'
            META_IDENTIFIER_TOKEN {store_intermediate_name(yytext);}
            '"' ';'
            same_definition
          ;
same_definition
          : WHEN_SAME_TOKEN INCOMPLETE_DEFINITION_TOKEN ';'
          | /* null */
          ;
data_items_passed_from_parser
          : DATA_ITEMS_PASSED_FROM_PARSER_TOKEN  ';' lines END_TOKEN ';'
          ;
lines
          : item_line
          | lines item_line
          ;
item_line
          : '<' CHAR_TOKEN              {store_data_type(yytext);}
            '*' '>' IDENTIFIER_TOKEN    {store_type_id(yytext);}
            ';'
          | '<' INT_TOKEN               {store_data_type(yytext);}
            '>' IDENTIFIER_TOKEN        {store_type_id(yytext);}
            ';'
          ;
semantic_routines
          : SEMANTIC_ROUTINES_TOKEN  ';' semantic_routine END_TOKEN ';'
semantic_routine
          : name_and_number_definition
          | semantic_routine name_and_number_definition
          ;
name_and_number_definition
          : '[' INTEGER_TOKEN ']'
            IDENTIFIER_TOKEN        {store_in_routines(yytext);}
            '('                     {store_in_routines(yytext);}
            parameters
            ')'                     {store_in_routines2(yytext);}
            ';'
          ;
parameters
          : parameter
          | parameters ','   {store_in_routines(yytext);}
            parameter
          ;
```

```
parameter
          : CHARACTER_TOKEN        {store_in_routines(yytext);}
          | IDENTIFIER_TOKEN       {store_in_routines(yytext);}
          | /* no parameters  */
          ;
semantic_routine_usage
          : SEMANTIC_ROUTINE_USAGE_TOKEN ';' lines END_TOKEN ';'
          ;
lines
          : line
          | lines line
          ;
line
          : '<' IDENTIFIER_TOKEN    {store_in_routine_usage(yytext);}
            '>'  USES_SEMANTIC_ROUTINES_TOKEN
            '[' routine_numbers ']'
          ;
routine_numbers
          : INTEGER_TOKEN  {store_in_routine_usage(yytext);}
          | routine_numbers ','
            INTEGER_TOKEN  {store_in_routine_usage(yytext);}
          ;
text_buffer_usage
          :  TEXT_BUFFER_USAGE_TOKEN ';' text_buffer_lines END_TOKEN ';'
          ;
text_buffer_lines
          : IDENTIFIER_TOKEN  {store_in_text_buffer(yytext);}
            '(' ')' ';'
          | text_buffer_lines
            IDENTIFIER_TOKEN  {store_in_text_buffer(yytext);}
            '(' ')' ';'
          ;
object_structures
          : CROSS_REFERENCER_OBJECT_STRUCTURES_TOKEN ';'
            structure_definitions
            END_TOKEN ';'
          ;
structure_definitions
          : structure_definition
          | structure_definitions structure_definition
          ;
structure_definition
          : '<' IDENTIFIER_TOKEN {store_in_object_structure(yytext);}
            '>' SLOT_STRUCTURE_IS_TOKEN
            '('       {store_in_object_structure(yytext);}
            slots ')' {store_in_object_structure(yytext);}
          ;
slots
          : IDENTIFIER_TOKEN  {store_in_object_structure(yytext);}
```

```
          | slots ','
            IDENTIFIER_TOKEN {store_in_object_structure(yytext);}
          ;
object_slot_descriptions
          : OBJECT_SLOT_DESCRIPTIONS_TOKEN ';' slot_descriptions
            END_TOKEN ';'
          ;
slot_descriptions
          : slot
          | slot_descriptions slot
          ;
slot
          : IDENTIFIER_TOKEN {store_in_slot_type(yytext);}
            ':'  TYPE_ID_TOKEN  IS_TOKEN   first_part
            TYPE_TOKEN IS_TOKEN type_part
            SLOT_VALUE_TOKEN IS_TOKEN   second_part
            META_SLOT_VALUE_TOKEN IS_TOKEN
            META_IDENTIFIER_TOKEN  {store_in_slot_type(yytext);}
            END_TOKEN IDENTIFIER_TOKEN { store_in_slot_imp_type(yytext);}
          ;
first_part
          : IDENTIFIER_TOKEN {store_in_slot_type(yytext);}
          ;
type_part
          : CHAR_TOKEN { store_in_slot_imp_type(yytext); }
          | INT_TOKEN  { store_in_slot_imp_type(yytext); }
          ;
second_part
          : SINGLE_TOKEN  {store_in_slot_type(yytext);}
          | MULTI_TOKEN   {store_in_slot_type(yytext);}
          ;
source_identifiers_to_be_ignored   /* reserved identifiers */
          : SOURCE_CODE_IDENTIFIERS_TO_BE_IGNORED_TOKEN ';' identifier_lines
            END_TOKEN ';'
          ;
identifier_lines
          : IDENTIFIER_TOKEN  {store_reserved_id(yytext);}
            ';'
          | identifier_lines
            IDENTIFIER_TOKEN {store_reserved_id(yytext);}
            ';'
          ;
xref_listing_specification
          : LISTING_SPECIFICATION_TOKEN ';'
            terse_spec intermediate_spec END_TOKEN ';'
          ;
terse_spec
          : TERSE_SPEC_TOKEN ';' terse_packets END_TOKEN ';'
          ;
```

```
intermediate_spec
          : INTERMEDIATE_SPEC_TOKEN ';'  int_packets END_TOKEN ';'
          ;
terse_packets
          : IDENTIFIER_TOKEN {store_terse_spec(yytext);}
            ';'
          | terse_packets  IDENTIFIER_TOKEN {store_terse_spec(yytext);}
            ';'
          ;
int_packets
          : IDENTIFIER_TOKEN  {store_intermediate_spec(yytext);}
            ';'
          | int_packets
            IDENTIFIER_TOKEN {store_intermediate_spec(yytext);}
            ';'
          ;
end_of_spec_token
          : END_OF_CROSS_REFERENCER_SPECIFICATION_TOKEN
          ;
```

Appendix F

A Symbol Table Template

```
macrolist1
 /*
  *   cross referencer and symbol table structure
  */
struct symbol_table
                   {
                    struct scope * outer_scope;  /* level 0 */
                   }
/* scope structure */
 struct scope
{
struct <<NEXT>> <OBJECT_CLASSES>_section* <<SAME>> <OBJECT_CLASSES>
struct scope * next;  /* order of scopes */ };
/* scope sections , the different classes of identifier */
struct <<SAME>> <OBJECT_CLASSES>_section
{
struct <<SAME>> <OBJECT_CLASSES>_object * <<SAME>> <OBJECT_CLASSES> ;
struct <<SAME>> <OBJECT_CLASSES>_section * left;
struct <<SAME>> <OBJECT_CLASSES>_section * right;
};
/* objects , a different object description for each type of symbol */
struct <<NEXT>> <OBJECT_CLASSES>_object
 {
 <<NEXT>> <OBJECT_DETAILS> ;
 };
struct linenumbers
```

```
  { /* chain of linenumbers */
int source_line_number;
 struct linenumbers * next_line_number;
  }
```

Appendix G

A Generated Symbol Table

```
 /*
  *   cross referencer and symbol table structure
  */
struct symbol_table
                  {
                   struct scope * outer_scope;  /* level 0 */
                  };
/* scope structure */
 struct scope
{
struct  parameter_class_section *  parameter_class_ptr ;
struct  label_class_section *  label_class_ptr ;
struct  constant_class_section *  constant_class_ptr ;
struct  type_class_section *  type_class_ptr ;
struct  var_class_section *  var_class_ptr ;
struct  function_class_section *  function_class_ptr ;
struct  procedure_class_section *  procedure_class_ptr ;
struct scope * next;  /* order of scopes */ };
/* scope sections , the different classes of identifier */
struct  parameter_class_section

{

struct  parameter_class_object *  parameter_class_ptr ;

struct  parameter_class_section * left_ptr;
```

```
struct  parameter_class_section * right_ptr;
};
struct  label_class_section
{
struct  label_class_object *  label_class_ptr ;
struct  label_class_section * left_ptr;
struct  label_class_section * right_ptr;
};
struct  constant_class_section
{
struct  constant_class_object *  constant_class_ptr ;
struct  constant_class_section * left_ptr;
struct  constant_class_section * right_ptr;
};
struct  type_class_section
{
struct  type_class_object *  type_class_ptr ;
struct  type_class_section * left_ptr;
struct  type_class_section * right_ptr;
};
struct  var_class_section
{
struct  var_class_object *  var_class_ptr ;
struct  var_class_section * left_ptr;
struct  var_class_section * right_ptr;
```

```
};

struct  function_class_section

{

struct  function_class_object *  function_class_ptr ;

struct  function_class_section * left_ptr;

struct  function_class_section * right_ptr;

};

struct  procedure_class_section

{

struct  procedure_class_object *  procedure_class_ptr ;

struct  procedure_class_section * left_ptr;

struct  procedure_class_section * right_ptr;

};

/* objects , a different object description for each type of symbol */
struct  parameter_class_object
 {
  char   object[256];
  char   namespace[256];
  char   class[256];
  char   pascal_type[256];
  struct linenumbers *  parameter_on;
  struct linenumbers *  inwith;
 };
struct  label_class_object
 {
  char   object[256];
  char   namespace[256];
  char   class[256];
  struct linenumbers *  declared;
  struct linenumbers *  undefined;
  struct linenumbers *  ingoto;
 };
struct  constant_class_object
 {
  char   object[256];
```

```
  char   namespace[256];
  char   class[256];
  char   const_value[256];
  struct linenumbers *  declared;
  struct linenumbers *  used;
 };
struct  type_class_object
 {
  char   object[256];
  char   namespace[256];
  char   class[256];
  char   pascal_type[256];
  struct linenumbers *  declared;
  struct linenumbers *  used;
 };
struct  var_class_object
 {
  char   object[256];
  char   namespace[256];
  char   class[256];
  char   pascal_type[256];
  struct linenumbers *  declared;
  struct linenumbers *  used;
  struct linenumbers *  var_as_parameter;
  struct linenumbers *  inwith;
  struct linenumbers *  infor;
 };
struct  function_class_object
 {
  char   object[256];
  char   namespace[256];
  char   class[256];
  struct linenumbers *  parameter_on;
  struct linenumbers *  external;
  struct linenumbers *  forward;
  char   non_local[256];
  struct linenumbers *  called;
 };
struct  procedure_class_object
 {
  char   object[256];
  char   namespace[256];
  char   class[256];
  struct linenumbers *  parameter_on;
  struct linenumbers *  external;
  struct linenumbers *  forward;
  char   non_local[256];
  struct linenumbers *  called;
  char   function_return_type[256];
```

```
 };
struct linenumbers
  { /* chain of linenumbers */
int source_line_number;
 struct linenumbers * next_line_number_ptr;
  };
```

Bibliography

[1] A.V. Aho, R. Sethi, J.D. Ullman, 1986, **Compilers Principles, Techniques and Tools**, *Addison-Wesley*

[2] IEEE Standards Board and ANSII Standards Institute, 1983, **An American National Standard and IEEE Standard Glossary of Software Engineering Terminology**, *ANSI/IEEE Std 729-1983* pp32

[3] American National Standard, 1989, **Programming Language Cobol**, *ANSI X3.23-1985, ISO 1989-1985*

[4] K.M. Broadey, M. Munro and D.J. Robson, School of Engineering and Applied Science, Computer Science, University of Durham, 1986, **A Context Sensitive Cross Reference Tool**, *Report 86/1*

[5] K.M. Broadey, A. Colbrook, M. Munro and D.J. Robson, 1989, **Block Structured Cross Referencer for Pascal and C**, *University Computing, 11(3)* pp120-128

[6] K.M. Broadey, School of Engineering and Applied Science, Computer Science, University of Durham, 1985, **A Context Sensitive Cross Referencer for the PASCAL Programming Language**, *Project Report*

[7] R.A. Brooker and D. Morris, 1962, **A General Translation Program for Phrase Structure Languages**, *J. ACM 9* pp1-10

[8] L.Chik, School of Engineering and Applied Science, Computer Science, University of Durham,, 1988, **A Replacement C Preprocessor Front End for the CXR Cross Referencer**, *Project Report*

[9] J.C. Cleveland, 1988, **Building Applications Generators**, *IEEE Software, 5(4)* pp25-33

[10] A. Colbrook, School of Engineering and Applied Science, Computer Science, University of Durham, 1987, **A Context Sensitive Cross Reference Tool for the Language C**, *Project Report*

[11] S.D. Cooper, School of Engineering and Applied Science, Computer Science, University of Durham, 1987, **Pascal Program Call Graph Generator**, *Project Report*

[12] R. Fairley, 1985, **Software Engineering Concepts**, *McGraw-Hill International Editions, Computer Science Series*

[13] J. Feldman and D. Gries, 1968, **Translator Writing Systems**, *Communications of the ACM, 11(2)* pp77-113

[14] G. Fischer, 1989, **Human-Computer Interaction Software: Lessons Learned, Challenges Ahead**, *IEEE Software, 6(1)* pp45-52

[15] N. Fletton, 1988, **M.Sc. Thesis, Documentation for Software Maintenance and the Redocumentation of Existing Systems**, *School of Engineering and Applied Science, Computer Science, University of Durham*

[16] J. Foley, W.C. Kim, S. Kovacevic, and K. Murray, 1989, **Defining Interfaces at a high level of abstraction**, *IEEE Software, 6(1)* pp25-36

[17] J.R. Foster, 1986, **Software Maintenance - An Overview Divisional Memorandum R11/86/013**, *British Telecom Research Laboratories*

[18] D. Gries, 1971, **Compiler Construction for Digital Computers**, *John Wiley Sons,Inc*

[19] L. Hancock and M. Kriege, 1987, **The C Primer Second Edition**, *McGraw-Hill Book Company*

[20] R. Hartson, 1989, **User Interface Management Control and Communication**, *IEEE Software, 6(1)* pp62-70

[21] K. Koskimies, O. Nurmi and J. Paakki, 1988, **The Design of a Language Processor Generator**, *Software Practice and Experience, 18(2)* pp107-135

[22] G. Lay, School of Engineering and Applied Science, Computer Science, University of Durham, 1987, **The Intermediate Representation of Cross Referenced Information with regard to output linkage**, *Vacation Report*

[23] G. Lay, School of Engineering and Applied Science, Computer Science, University of Durham, 1988, **The Intermediate Representation of Cross Referenced Information**, *Project Report*

[24] B.P. Lientz and E. Burton Swanson, 1980, **Software Maintenance Management**, *Addison Wesley*

[25] F.J. Luckey, 1989, **Understanding and Debugging Programs**, *Journal of Man Machine Studies 12* pp 202

[26] C. Marcus, 1986, **Prolog Programming**, *Addison Wesley*

[27] W.M. McKeeman, J.J. Horning, D.B. Wortman, 1970, **A Compiler Generator**, *Prentice-Hall Series in Automatic Computation*

[28] M. Munro and D.J. Robson, 1987, **An Interactive Cross Reference Tool for use in Software Maintenance**, *Proceedings of the 20th Annual Conference on Systems Science* pp64-70

[29] B.A. Myers, Carnegie Mellon University, 1989, **User Interface Tools: Introduction and Survey**, *IEEE Software, 6(1)* pp15-24

[30] P. Naur, 1963, **Revised Report on the algorithmic language Algol 60**, *Comm. of the ACM 6(1)* pp 1-17

[31] R.J. Pooley, 1988, **An Introduction to Programming in SIMULA**, *Blackwell Scientific Publications*

[32] I.C. Pyle, 1985, **The Ada Programming Language**, *Prentice Hall International*

[33] S.P. Reiss, 1983, **Generation of Compiler symbol processing mechanisms from specifications**, *ACM Trans.Programming Lang. Syst., 5(2)* pp127-163

[34] S.P. Reiss, 1987, **Automatic Compiler Production: The Front End**, *IEEE Transactions on Software Engineering, 13(6)* pp607-627

[35] H. Ruston, 1978, **Programming with PL/1**, *Addison Wesley*

[36] A.T. Schreiner, 1985, **An Introduction to Compiler Construction with UNIX**, *Prentice Hall, Inc.*

[37] C.H. Smedema, P. Medema, M.Boasson, 1983, **The Programming Language Pascal, Modula-2, CHILL and Ada**, *Prentice Hall, Inc.*

[38] E. Soloway and W.L. Johnson, 1980, **Knowledged Based Program Understanding**, *IEEE Trans. Software Eng. 11(3)* pp265-275

[39] H.M. Sneed, 1988, **Software Renewal: A Case Study**, *IEEE Software, 1(3)* pp56-63

[40] B. Stroustrup, 1986, **The C++ Programming Language**, *Addison Wesley*

[41] H. Thimbleby, 1988, **Delaying Commitment**, *IEEE Software, 5(3)* pp78-86

[42] A.B. Tucker, Jr., 1986, **Programming Languages**, *McGraw-Hill*

[43] J. Welsh and J. Elder, 1988, **Introduction to Programming in Modula-2**, *Prentice Hall*

[44] B.A. Wichmann, 1973, **ALGOL 60 Compilation and Assessment**, *Academic Press London and New York*

[45] N. Wilde, 1989, **The Maintenance Assistant Work in Progress**, *Journal of Systems and Software, 9(1)* pp3-18

[46] S. Williams, 1985, **Programming the 68000**, *SYBEX*

[47] L.B. Wilson and R.G. Clark, 1987, **Comparative Programming Languages**, *Addison Wesley*