In commenting on the cross section of computer science research in South Africa, I will use the classification in the table of contents of the "Summary of Awards: Fiscal Year 1989," a document published recently by the US National Science Foundation. Of the 5 categories, I will treat Numeric and Symbolic Computation as inappropriate for the discussion below. In this category I noted no research in the computer science setting in South Africa. It is also common in the US and elsewhere to place this effort in other departments, e.g., departments of mathematics or applied mathematics.

Of the remaining categories I found South Africa to be strongest in software systems and engineering, to have a substantial investment in computer systems and architecture, and to be weakest in computer and computation theory.

The coverage in software systems and engineering (SSE) was broad, topical, and similar in scope to that in US universities. Technology transfer and the corresponding relations with industry seemed to be in place or developing along promising lines. I comment in passing that this was rather surprising to me. In the US the development of SSE within university departments has lagged behind almost all other disciplines of computer science. A primary problem has been the insatiable appetite of industry for all Ph.D. graduates in the SSE field.

The investment in parallel processing, computer networks, and distributed computing appears sound, although I expected to see a greater emphasis on mathematical foundations (see my remarks below), particularly in the parallel algorithms area. Given current resources, South African institutions are doing remarkably well in computer science research. But computer science is a fundamentally important course of study, beginning at an early age and extending through graduate Ph.D. research; I take this as sufficiently obvious that I need not dwell on justifications. With this in mind, and with the necessary resources in hand, South Africa should, in my opinion, expand and consolidate its computer science research effort, increase its visibility in the international arena, and correct the rather thin distribution of graduate research among universities.

I can see much of this proceeding along present lines, but I would strongly recommend a concerted development in computer and computation theory (CCT), education and research; this is mainstream computer science and forms the basis for virtually all other fields of study within computer science. It is by no means absent in South Africa curricula, but it appears to be under-represented in advanced studies and Ph.D. level research.

At the graduate level CCT is heavily mathematical. I understand that mathematical foundations are supplied by mathematics departments in certain cases. This is not ideal, but workable and it is justified by limited resources. However, it is important that mathematics departments not regard this as a mere service; faculty will have to make a major commitment to theoretical computer science, publishing in its leading journals (e.g. SIAM Journal of Computing, Journal of the ACM, Journal of Algorithms, Algorithmica, Journal of Computer and Systems Sciences, Theoretical Computer Science, etc.), and providing the supervision of theses sponsored by computer science departments and leading to degrees in computer science. I would also encourage active participation in the international computer science "theory" societies and their meetings; two highly prestigious examples of the latter are the annual Symposium on the Theory of Computing and the Foundations of Computer Science conference.

Returning to the thin distribution of computer science research, I would make the following point. If the current situation is only a stage of development - i.e., if further resources (both human and financial) can be counted on to bring at least a few of the departments to a critical mass - then little needs to be said beyond the earlier remarks. Critical mass is hard to define, but calls for adequate, expert coverage of mainstream computer science research. In view of the breadth of this research, 8-10 Ph.D. full-time-equivalent faculty would seem to be barely adequate; with the usual clumping of faculty in specific research areas, more would be expected. South Africa has a talent base such that there is little doubt that such departments would achieve a much wider international recognition.
On the other hand, if resources remain fixed at current or even slightly retrenched levels, then I would recommend consolidation to achieve the same goals on a smaller scale. Within a university, this can often be done by establishing interdisciplinary, degree-granting laboratories or institutes of computer science, which bring together the computer science efforts located in various departments other than computer science, such as electrical engineering, industrial engineering, business-/management science, mathematics, and operations research. The idea is to enjoy the advantages (opportunity, synergy, awareness, etc.) to both students and faculty of reasonably large computer science programs. There are many examples of such intramural laboratories in North America and Europe.

This approach could also be considered among

**Editor's Notes**

Prof John Schochot has graciously accepted to be SACJ’s subeditor for papers relating to Information Systems. Authors wishing to submit papers in this general area should please contact him directly. I look forward working with John, and to a significant increase in IS contributions in future.

The hand of the new production editor, Riel Smit, will be clearly evident in this issue. Those papers not prepared in camera-ready format by the authors themselves were prepared by him in TEX. He will be announcing revised guidelines for camera-ready format in a future issue. If you use TEX or one of its variations, Riel would be happy to provide you with a styles document to SACJ format.

At last some Department of National Education committee has decided that SACJ should now be on the list of approved journals. This places it amongst the ranks of some 6800 other journals. These include not merely a number of ACM and IEEE Transactions but also such journals as Ostrich, Trivium, Crane Bag, Koers, Mosquito News, Police Chief, Connoisseur, Lion and the Unicorn, About the House and Ohio Agricultural Research and Development Center Department Series ESS. You will recall that in 1990 this same committee decided that, if judged on its own merits, SACJ did not deserve to be on the illustrious list. In the absence of other evidence, we must assume that the sole reason for its revised decision is that SACJ’s predecessor, Questiones Informaticae, was there. (I have a secret suspicion that the committee liked that name.) It is my understanding that for official purposes, all journals on this list are regarded as *equally* meritorious, and all of them are more meritorious than *any* conference proceedings. What does all of this mean?

The momentous implication of the committee’s deliberations is that the State will not give your institution a single cent for anything that you publish in SACJ. Instead, the State and your institution will scrupulously keep a score of the annual number of publications that count - but actually don’t - because someday they might! And to encourage your enthusiastic participation in this Alice in Wonderland exercise, your institution might actually give you some of the standard subsidy funding that the State should have provided according to its own formulae, but didn’t.

You will not be allowed to use this money to buy yourself a car - not even a casual meal. You may only use it to finance activities that are provably directed towards producing more papers. The great consolation, of course, is that you will not be required to pay income tax on this money. The only tax involved will be the VAT component when you spend it in an approved manner. As a good computer scientist who enjoys recursion, my vote would be that all such revenue collected by the State should be earmarked to be placed in the pay packets of committee members who decided that SACJ should be approved.

If you publish in these approved journals with sufficient regularity and enthusiasm you will almost deserve to be regarded as a researcher. What you additionally need to do, is to ensure that you befriend and impress at least three overseas referees. You then apply to the FRD for official recognition as a researcher, and if they are sufficiently impressed, they will give you more of the non-taxable kind of money that you need to spend on research to publish in approved journals.

Derrick Kourie
Editor
Abstract
The parallel conditional, a natural generalisation of both Lisp's COND and Dijkstra's guarded commands, is particularly suited to parallel processing. This note describes the implementation of a preprocessor that enables use of the parallel conditional in C on a multiprocessor machine, the Sequent Balance 8000.

Keywords: language construct, conditional, guarded command


Received September 1991, Accepted November 1991

1. Introduction
A conditional is a language construct that consists of a sequence $P_1 \ldots P_n$ of guards and a sequence $X_1 \ldots X_n$ of corresponding blocks, and syntax and semantics rules for the construct. The formal semantics of the parallel conditional, and its relationship to the COND conditional of Lisp [I] and to Dijkstra's guarded commands [2], was investigated in [3]. The of a conditional is to select one of the blocks and its operational purpose semantics can be described by a rule that specifies how it does this selection in terms of an evaluation strategy applied to the guards. The operational semantics of the parallel conditional is particularly simple - it selects non-deterministically an $X_i$ whose corresponding $P_i$ evaluates to true. The non-determinism here is the don't care variety. Operationally it is appropriate to evaluate the guards in parallel and to select the first $X_i$ for which the corresponding $P_i$ evaluates to true. The evaluation of the remaining $P_i$ can then be terminated. It is not necessary that all the $P_i$ be defined. (In practice $P_i$ is undefined if its evaluation is a non-terminating process or produces a runtime error.) The parallel conditional is itself undefined if there is no $P_i$ that evaluates to true.

We outline the construction of a preprocessor for C programs with embedded parallel conditionals. The target machine is the Sequent Balance. This is a multiprocessor machine whose operating system, Dynix BSD 4.2, is an extension of Unix that will run a new process on a new processor (if one is free).

2. The preprocessor
The programmer is allowed to embed parallel conditionals in a C program using the syntax $\langle P_1=X_1, \ldots, P_n=X_n \rangle$. Each $P_i$ and each $X_i$ is a C function call. The preprocessor modifies a source program so that the modified source compiles to code that evaluates the guards concurrently and executes a block whose guard is found to be true. Communication and control is by means of a simple kind of semaphore - a process can lock an unlocked share variable but will suspend if it attempts to lock a locked variable. The C language supplied for the machine is an extended C that provides this mechanism. The preprocessor modifies each function in the original program that contains a parallel conditional, producing a new function with the following structure:

```
lock(control); < control is a shared variable
< modified version of original body
kill all child processes;
```

The housework referred to above consists mainly of declarations and of code to maintain dynamically a pool of child processes. Each time a parallel conditional is encountered child processes are allocated from the pool, one per guard. New child processes are created if the pool is too small. The modified body of the original function is obtained by the preprocessor by substituting the following for each parallel conditional statement:

```
guardnumber:=0;  < guardnumber is to become i where the i'th guard evaluates to true.
< guardnumber is shared)
construct a list L of guards; < the list L is shared with child processes
create child processes (if needed);
unlock(control); < let child processes evaluate guards
while guardnumber=0 wait; < for a true guard
lock(control); < to suspend child processes after they are restarted
restart all child processes; < and pool them
execute X_i where i=guardnumber;
```
The child processes are all identical:

forever do
    lock(control);  // wait if control is locked
    if guardlist L not eq:=ty then
        // child has exclusive access to L at this point
        get a guard P1;
        // take a guard P1 from L
        unlock(control);
        // let another child process proceed
        evaluate P1,
        if true then guardnumber:=i;
        else unlock(control);
        wait;  // to be restarted

3. Limitations and user aspects

To keep the initial implementation simple, parallel conditional statements cannot be nested. A common problem with preprocessors is that they impose additional restrictions on the programmer - for example in our case there are 17 identifiers that must be reserved for use by the preprocessor. A description of these is contained in a user manual that also describes the syntax and semantics of the conditional. Most importantly, the manual advises the user on debugging source programs that contain the conditional. We had to actively discourage the user from hacking the C code produced by the preprocessor. He wanted to do this when his program did not 'work', since he was tempted to believe that his own programming must be correct and that in some way the preprocessor had ' messed up'. We therefore had to include in the manual some elementary advice about using test data and advising him to replace a whole parallel conditional by one of its blocks for test purposes.

The semantics of the parallel conditional causes a particular problem with debugging that led us to include a special debugging tool for it in the preprocessor. The conditional is undefined if no guard is true and our implementation produces a non-terminating program in this case. It is unreasonable to expect the programmer to always be able to determine in advance that at least one of his guards becomes true. Indeed this is undecidable in general. Therefore it should be possible to terminate a program cleanly if it is suspected that it is stuck in an undefined conditional. Killing the program externally is unsatisfactory since this can leave child processes executing and these not easily identified among the many processes that execute concurrently on a complex machine. To alleviate the problem, we added a special user function inform error() to the preprocessor. The manual encourages the programmer to include an extra clause time out() ⇒ inform error() in each conditional while debugging. The suggested function for time out is:

```c
int time_out()
    { sleep(<seconds>);
      return(1) }
```

The action of inform error() is to
a) inform the user that only time_out has returned true,
b) kill all child processes that are evaluating guards, and
c) kill the user program.

4. User experience

We believe that providing only low level mechanisms in an extended C for parallel programming does not facilitate the production of reliable software. Short of writing a new compiler, the preprocessor seemed a reasonable way to provide a powerful parallel programming construct at a high level. Implementation details are hidden from the user by using a shell to preprocess and compile.

We had hoped that explaining the parallel conditional to graduate students, and making the preprocessor and its manual available to them, would encourage use of the conditional and lead to some interesting applications. We pointed out that the parallel conditional can be used as a general mechanism for launching or-parallel processes, for example the or-parallel execution of P1,P2,P3 corresponds to P1→X1, P2→X2, P3→X3 where each X_i simply returns true.

To our disappointment, the preprocessor has been little used. While students express interest and think the preprocessor ' cute', they prefer to work with the low level control constructs already provided in C for the Balance. They give two reasons for this. Firstly, (in common with many C programmers) they find working at a low level ' fun'. Secondly, and more reasonably in our view, they find that they need to be able to use the existing low level process creation and control mechanisms for other purposes, so that using the parallel conditional and the preprocessor reduces their practice and experience with these mechanisms.

References

Book Reviews


Reviewer: Professor J Bishop, University of Pretoria (This review was also commissioned by the Computer Journal and will appear therein during 1992.)

Any book that comes out in a third edition in the cut-throat world of first year Pascal texts, deserves close scrutiny. These two books are a sample from a suite of six books, two laboratory courses and a video course that Nell Dale and colleagues have published through D C Heath since 1983. They have evidently refined their winning formula to the point where it is a "success at over 1250 schools" (quoted from the publisher's advertising). So what is it that the customers like? Why are these books so successful?

There seem to be two factors involved: pedagogy and quality. Both books cover their topics absolutely thoroughly, explaining each new feature as it arises, using interesting examples, both classic (binary search) and novel (absenteeism pattern). Some of the examples progress through the book, becoming more sophisticated as the student's expertise increases, and there are several larger case studies. The order of topics has been carefully chosen, and the authors have adapted the material through each revision so as to reflect changing trends. The teaching and learning aids are impressive: the introductory book concludes each chapter with case studies, advice on testing and debugging, and then four levels of exercises for the students, from a quick check quiz to genuine programming problems. Beginners will find these very reassuring. The advanced book includes a diskette containing all the programs, which is a very good sign that the programs have been well tested. Both books are produced in two colours, with cartoons and many graphic illustrations. Quality indeed!

The introductory book has 17 chapters, which take a genuine beginner through the programming process, design methodology and problem solving, to simple Pascal with control structures, procedures and parameters, on to functions, data types and recursion. Purists can note that looping is taught with while statements and that repeat and for only appear later on in the data types section. Subranges are emphasised from chapter 10 onwards, and I was glad to see that they come before arrays. Unlike many books that dive into array handling at the start, Nell and Weems take this conceptually difficult hurdle slowly. Chapters 11, 12 and 13 introduce array processing, patterns of array access, lists, strings and tables. Chapter 15 covers files and pointers and chapter 16 provides a gentle introduction to the advanced book on data structures.

The only problem with this book is that it covers standard Pascal and only standard Pascal. Thus many common operations (reading in a string) are long-winded, some (e.g. assigning a file name) have to be side-stepped, and the important programming issues of screen and graphics handling, separate compilation and objects, are simply ignored. There is a companion volume which is a Turbo Pascal version, and one hopes that these aspects are fully addressed there.

Because the advanced book follows the same easily accessible style and high-quality layout of the introductory text (complete with jokes), I was tempted to regard it as not a serious contender in the Data Structures and Algorithms text stakes. Closer scrutiny proved me wrong. Dale and Lilly have managed to retain a high level of rigour in the presentation of abstract data types while still making each of them truly usable in practice. Stacks, queues, lists and binary trees are covered in detail, and the efficiency of each of the algorithms applied to them is formally discussed. Sorting and searching (including hashing) receive similar treatment. However, the book stops short of the next level of ADTs - bags, sets, B-Trees, directed graphs. Turbo Pascal is used where the authors were really high standard, using plentiful comments, good type definitions, and procedures with parameters to the full. The advanced book includes a diskette containing all the programs, which is a very good sign that the programs have been well tested. Both books are produced in two colours, with cartoons and many graphic illustrations. Quality indeed!
desperate, and there is also a Turbo Pascal version of
the book (but how "Turbo" it is, I do not know).

So, how would these books fit into first courses
outside the US? The introductory book is very basic
and very slow. The insight into programming is fine, but is
dispersed throughout the text, and overall the book is
just too big for the subject it covers. On the other hand,
the pedagogical aids are first-class and I would certainly
advise any student who is having difficulty with a Pascal
course, to get this book as back-up. The advanced book
is also hefty, and although thorough, does not go as far as
into the exciting aspects of algorithms as the more
formal and classic texts do. On the other hand, the
examples and assignments are first rate, and I am very
glad to have this book on my shelf.

These are both excellent books in all respects except
coverage, and I would certainly recommend that any
teachers of first or second year should inspect copies and
judge whether the coverage is sufficient for the courses
under consideration. If it is, then both class and lecturer
will be joining the other 1250 satisfied schools, and are
on to a winner.

---

Introduction to the Theory of Programming
Languages, by B Meyer, Prentice Hall, 1990, 447pp,
ISBN 0 13 498502 8
Reviewer: DG Kourie, University of Pretoria

Someone who has designed an elegant programming
language might legitimately be expected to know about
the theory of programming languages. As to the first
part of this conjecture, the author of the book under
review, Bertrand Meyer, is also the designer of Eiffel,
an object oriented programming language that appears to
have impressed those who have had the opportunity of
studying and using it. Having used his book for the
second successive year in a programming language
course, I consider myself reasonably well-equipped to
comment on the second part of the opening conjecture.
The book provides ample evidence not only of Meyer's
personal theoretical grounding, but also of his
commitment and ability to keep theory and practice as
close together as possible. This aspect of the book (and
I suspect of Eiffel as well) has particular appeal.

The book is about ways to specify the syntax and
semantics of computer languages. The thesis is that
when you design a computer language, you do well to
initially abstract away from details of concrete syntax,
and you need to think deeply about the meaning of the
various constructs that you provide. These remarks
apply even if the language at issue is as mundane as a
user-interface. Consequently, the author includes
software engineers and pragmatic (as opposed to
theoretical) computer scientists in his targeted reader
group. His declared aim is to convince at least "some of
them that the mathematical basis of programming is
simple, useful and even pleasurable". Three sub-themes steer the book: abstract syntax,
denotational semantics and axiomatic semantics.
Although the book is self-contained in terms of
formal mathematical prerequisites, a certain level of
mathematical maturity will be necessary to assimilate
both the material. Chapter 2 on mathematical
preliminaries (mostly function theory and notation),
chapter 5 on lambda calculus and chapter 8 on the
mathematics of recursion provide all the background
and notation that the reader will need. Material in the
latter two chapters that is not referred to
subsequently, is clearly marked as "may be skipped
on first reading". The author views chapter 8 as an
exorcism", to justify the frequent use of recursion not
only in the theoretical derivations of the book but in
programming exercises in general. While interesting,
the chapter is not entirely essential for an
understanding of the book's main themes.

For illustrative purposes, the author uses a small
language that contains the typical core instructions
common to most procedural languages. Meyer calls
this language Graal, and I will not steal his thunder
by revealing the origin of the acronym. In chapter 3,
the abstract syntax of Graal is provided. Since it is
used so frequently throughout the remainder of the
book it is repeated on the back inside cover for easy
reference.

Chapter 4 gives a comparative overview of five
approaches to semantic specification: attribute
grammars, followed by translational, operational,
denotational and axiomatic semantics. A short
introductory course on syntax and semantics could
well be based on these first four chapters alone.

However, the real challenge (and fun) comes in
later chapters. Chapter 6 provides a detailed
exposition of the static and dynamic semantics of
Graal constructs. This implies a systematic derivation
of denotational functions to identify when constructs
are valid, and to express the meaning of valid
constructs. Whenever appropriate, the author takes
the trouble to point out how mathematical results
relate to practical concerns in language design.

The style of chapter 7 is much the same as 6. In
it, static and dynamic semantics are derived for
constructs which were not in the initial definition of
Graal. These include records, arrays, pointers, block
structure and routines.

Perhaps chapter 9, being about 100 pages in
length, should have been split into two or three
chapters. As it stands, it is too long to maintain a
focus. It is devoted entirely to axiomatic semantics,
providing and contrasting Floyd/Hoare semantics and
Dijkstra's weakest precondition semantics. The
chapter also discusses the constructive approach to
programming based on the weakest precondition
rules. Finally, chapter 10 shows how the denotational
and axiomatic theories inter-relate.

Most chapters are followed by exercises. The
group of honours students who have been using the
book have found these exercises to be very challenging. Nevertheless, there is an almost unanimous view from students themselves that assignments based on the exercises are useful, since they are forced to read the text more carefully and thereby assimilate results.

There is one feature of the book which forbids a wholehearted recommendation: it is riddled with typing errors. At one stage during the course, an assignment was given to find all the typing errors in a given chapter. By awarding one mark per error found, it was possible to score more than 20 marks for the assignment. These errors are a great pity. Confidence is so undermined that the reader is never too sure of the reason for not understanding a part of the text: has something been missed, or is the problem merely some typing error that has not been spotted? More generally, the presence of so many typing errors undermines one of the important messages that the book tries to convey, namely the value to be gained by the precision and rigor of formal methods in studying programming languages.

Notwithstanding these errors, I think that the book ultimately succeeds in its goal. It will persuade at least the more thoughtful amongst its targeted readership of the usefulness of formal methods. Some might even have the joy of finding them pleasurable and elegant. Few are likely to find them simple in the sense of being easy to understand on first reading.
Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal papers in Afrikaans will also be accepted.

Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for review should be prepared according to the following guidelines.
• Use double-space typing on one side only of A4 paper, and provide wide margins.
• The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
• Tables and figures should be on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.
• Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.
• References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets. References should thus take the following form:
Manuscripts accepted for publication should comply with the above guidelines, and may be provided in one of the following formats:
• in a typed form (i.e. suitable for scanning);
• as an ASCII file on diskette; or
• as a WordPerfect, \text{T\text{E}X} or \LaTeX\text{} file; or
• in camera-ready format.
  A page specification is available on request from the editor, for authors wishing to provide camera-ready copies. A styles file is available from the editor for Wordperfect, \text{T\text{E}X} or \LaTeX\text{} documents.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs
Proofs of accepted papers will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words.
Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.
Contents

GUEST CONTRIBUTION

Impressions of Computer Science Research In South Africa
E.G. Coffman, Jr. ......................................................... 1

RESEARCH ARTICLES

An Implementation of the Linda Tuple Space under the Helios Operating System

Modelling the Algebra of Weakest Preconditions
C Brink and I Rewitzky .................................................. 11

The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Oriented OS
KJ McGregor and RH Cambell .......................................... 21

An Object Oriented Framework for Optimistic Parallel Simulation on Shared-Memory Computers
P Machanik ................................................................. 27

Analysing Routing Strategies in Sporadic Networks
SW Melville .................................................................. 37

Using Statecharts to Design and Specify a Direct-Manipulation User Interface
L Van Zijl & D Mitton .................................................... 44

Extending Local Recovery Techniques for Distributed Databases
HL Viktor & MH Rennhackkamp ......................................... 59

Efficient Evaluation of Regular Path Programs
PT Wood ......................................................................... 67

Integrating Similarity-Based and Explanation-Based Learning
GD Oosthuizen & C Avenant ........................................... 72

Evaluating the Motivating Environment For IS Personnel in SA Compared to the USA. (Part I)
JD Cougar & DC Smith ...................................................... 79

TECHNICAL NOTE
An Implementation of the Parallel Conditional
U Jayasekera and NCK Philips ............................................ 85

COMMUNICATIONS AND REPORTS

Book Review ................................................................. 87
The CSP Notation and its Application to Parallel Processing
PG Clayton ................................................................. 90