SUID-AFRIKAANSE REKENAARSSIMPOSIUM
SOUTH AFRICAN COMPUTER SYMPOSIUM
HOLIDAY INN PRETORIA
JULIE 1 – 3 JULY 1987
Proceedings
of the
4th South African Computer Symposium

Holiday Inn, Pretoria
1 – 3 July 1987

edited by

Pieter Kritsinger
Computer Science Department
University of Cape Town
PREFACE

Computer science is an emerging discipline which is having difficulty in being recognised as a worthy member of the sciences. I will paraphrase John Hopcroft, co-winner of the 1986 Turing Award, when, during a recent interview, he said that the primary reason for the lack of recognition, is the age of our researchers. Probably not one of the researchers who presented their work at this symposium is older than 45. I know of no computer scientist in South Africa who is in a position where (s)he can affect funding priorities. As far as I know we have no representation on any of the committees of the Foundation for Research Development and for our Afrikaans speaking fraternity, none who is a member of the Akademie vir Wetenskap en Kuns. It will take time and conscious effort to establish our presence. The same is true of course for our universities. Again, with one exception, I know of no dean of a science faculty, vice-principal or principal who is a computer scientist. We consequently spend an enormous amount of time trying to explain the needs of computer science and its difficulties. I believe this symposium is a further step towards accreditation by our peers and superiors from the other sciences.

The total number of papers submitted to the Programme Committee for consideration was 34. Each paper was reviewed by three persons knowledgeable in the field it represents. Of those submitted, 23 were finally selected for inclusion in the symposium. As a result the overall quality of the papers is high and as a computer science community in Africa we can be justly proud of the final programme.

This is the fourth in the series of South African computer symposia. This year the symposium is sponsored by the Computer Society of South Africa (CSSA), the South African Institute for Computer Scientists and the local IFIP Committee. The executive director of the CSSA and his staff deserve warm thanks for handling the organisation as well as they have, while the Organising Committee provided Derrick and I with very valuable advice.

Finally I would like to express my sincere appreciation to the authors, to the members of the Programme Committee and particularly the reviewers. Without the kind cooperation of everyone, this symposium would not have taken place.

Pieter Kritzinger
SYMPOSIUM CHAIRMAN: PS Kritzinger, University of Cape Town.
SYMPOSIUM CO-CHAIRMAN: D Kourie, University of Pretoria.
MEMBERS OF THE PROGRAMME COMMITTEE

Judy Bishop, Witwatersrand University
Chris Bornman, UNISA
Hannes de Beer, Potchefstroom University
Gideon de Kock, Port Elizabeth University
Jaap Kies, Western Cape University
Derrick Kourie, Pretoria University
Pieter Kritzinger, Cape Town University
Tony Krzesinski, Stellenbosch University
Michael Laidlaw, Durban Westville University
Peter Lay, Cape Town University
Ken MacGregor, Cape Town University
Theo McDonald, Orange Free State University
Jan Oosthuizen, University of the North
Dennis Riordan, Rhodes University
Alan Sartori-Angus, Natal University
John Shochot, Witwatersrand University
Theuns Smith, Rand Afrikaans University
Trevor Turton, ISM (Pty) Ltd
Gerrit Wiechers, Infoplan.
LIST OF REVIEWERS

BERMAN Sonia
BISHOP Judy
BORNMAN Chris
CAREY Chris
CHERENACK Paul
DE BEER Hannes
DE VILLIERS Pieter
GORRINGE Pen
KIES Jaap
KOURIE Derrick
KRITZINGER Pieter
KRZESINSKI Tony
LAIDLAW Michael
LAY Peter

MacGREGOR Ken
MATTISON Keith
McDONALD Theo
RENNHACKKAMP Martin
RIORDAN Denis
SATORI-ANGUS Alan
SCHOCOT John
SMITH Theuns
TURTON Trevor
VAN DEN HEEVER Roelf
VAN ROOYEN Hester
VON SOLMS Basie
VOS Koos
TABLE OF CONTENTS

Keynote Address

"An Extensible System and Programming Tool for Workstation Computers." ........................................... 1
Niklaus Wirth, ETH, Zurich

Invited Lectures

"The Relationship of Natural and Artificial Intelligence." .............. not included in Proceedings.
G Lasker, University of Windsor, Ontario.

"Software Engineering: What Can We Expect in the Future?" ........... not included in Proceedings.
D Teichrow, University of Michigan, U.S.A.

Computer Languages I

"SPS-Algo·l: Semantic Constructs for a Persistent Programming Language." ................................. 13
S Berman, University of Cape Town.

"Petri Net Topologies for a Specification Language." .... 25
R Watson, University of the Witwatersrand.

"Towards a Programming Environment Standard in LISP." ... 45
R Mori, University of Cape Town

"ADA for Multiprocessors: Some Problems and Solutions." 63
J Bishop, University of the Witwatersrand.

Computer Graphics

"Polygon Shading on Vector Type Devices." ............ 75
C F Scheepers, CSIR.

"Hidden Surface Elimination in Raster Graphics Using Visigrams." ................................. 97
P Gorringe, CSIR.

Database Systems I

"On Syntax and Semantics Related to Incomplete Information Databases." ................................. 109
M E Orlowska, UNISA.

"Modelling Distributed Database Concurrency Control Overheads." .................................................. 131
M H Rennhackkamp, University of Stellenbosch.

Operating Systems

"The Development of a Fault Tolerant System for a Real-time Environment." ................................. 149
M Morris, CSIR.

"A New General-purpose Operating System." ................. 161
B H Venter, CSIR.
Computer Languages II

"The Representation of Chemical Structures by Random Context Structure Grammars." ......................... 175
E M Ehlers and B von Solms, RAU.

"A Generalised Expression Structure." ...................... 189
W van Biljon, CSIR.

Computer Networks and Protocols I

"An Approximate Solution Method for Multiclass Queueing Networks with State Dependent Routing and Window Row Control." ........................................ 203
A E Krzesinski, University of Stellenbosch.

"A Protocol Validation System." ........................... 227
J Punt, University of Cape Town.

Computer Networks and Protocols II

P S Kritzinger, University of Cape Town.

Artificial Intelligence

"A Data Structure for Exchanging Geographic Information." ......................................................... 267
A Cooper, CSIR.

"The Design and Use of a Prolog Trace Generator for CSP." ....................................................... 279
D G Kourie, University of Pretoria.

Database Systems II

"An Approach to Direct End-user Usage of Multiple Databases." .................................................. 297
M J Phillips, CSIR.

"A Semantic Data Model Approach to Logical Data Independence." ............................................... 329
S Berman, University of Cape Town.

Information Systems

"The ELSIM Language: an FSM-based Language for the ELSIM SEE." ............................................ 343
L du Plessis and C Bornman, UNISA.

"Three Packaging Rules for Information System Design." 363
J Mende, University of the Witwatersrand.
Computer Languages III

"Experience with a Pattern-matching Code Generator." ... 371
M A Mulders, D A Sewry and W R van Biljon, CSIR.

"Set-oriented Functional Style of Programming." ........ 385
C Mueller, University of the Witwatersrand.

Tutorial

The use of Modula-2 in Software Engineering." ............ 399
N Wirth, ETH, Zurich.
<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>07h30</td>
<td>Registration and Coffee.</td>
</tr>
<tr>
<td>08h45</td>
<td>Welcoming address, President of the South African Institute of Computer Scientists, Dr. G. Wiechers.</td>
</tr>
<tr>
<td>09h00</td>
<td>Invited Lecture. Professor D. Teichrow, University of Michigan. Software Engineering, ... What Can We Expect in the Future.</td>
</tr>
<tr>
<td>10h00</td>
<td>COFFEE</td>
</tr>
<tr>
<td>10h15</td>
<td>Computer Languages I. Chairman: G. Wiechers.</td>
</tr>
<tr>
<td>10h50</td>
<td>S. Berman, University of Cape Town. SPS-Algol: Semantic Constructs for a Persistent Programming Language.</td>
</tr>
<tr>
<td>10h50</td>
<td>A. Watson, University of the Witwatersrand. Petri Net Topologies for a Specification Language.</td>
</tr>
<tr>
<td>11h25</td>
<td>A. Mori, University of Cape Town. Towards a Programming Environment Standard in USP.</td>
</tr>
<tr>
<td>11h50</td>
<td>J. Bishop, University of the Witwatersrand. ADA for Multiprocessors: Some Problems and Solutions.</td>
</tr>
<tr>
<td>12h30</td>
<td>LUNCH</td>
</tr>
<tr>
<td>14h00</td>
<td>Computer Graphics. Chairman: D. Kourie</td>
</tr>
<tr>
<td>14h35</td>
<td>P. Gorringe, CSIR. Polygon Shading on Vector Type Devices.</td>
</tr>
<tr>
<td>15h15</td>
<td>COFFEE</td>
</tr>
<tr>
<td>15h30</td>
<td>Database Systems I. Chairman: B. von Solms.</td>
</tr>
<tr>
<td>16h05</td>
<td>M. H. Renhadkamp, Stellenbosch University. Modelling Distributed Database Concurrency Control Overheads</td>
</tr>
<tr>
<td>18h00</td>
<td>Cocktail Party in Cullinan Room A.</td>
</tr>
</tbody>
</table>
# Day 2

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>08h30</td>
<td>Keynote Address by Professor Niklaus Wirth, Swiss Federal Institute</td>
</tr>
<tr>
<td></td>
<td>for Technology, Zurich.</td>
</tr>
<tr>
<td></td>
<td><em>An Extensible System and a Programming Tool for Workstation Computers.</em></td>
</tr>
<tr>
<td></td>
<td><strong>Computer Networks and Protocols I.</strong> Chairman: P.S. Kritzinger.</td>
</tr>
<tr>
<td>09h30</td>
<td>A.E. Krzesinski, University of Stellenbosch.</td>
</tr>
<tr>
<td></td>
<td><em>An Approximate Solution Method for Multiclass Queueing Networks</em></td>
</tr>
<tr>
<td></td>
<td><em>with State Dependent Routing and Window Flow Control.</em></td>
</tr>
<tr>
<td>10h05</td>
<td>J. Punt, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><em>A Protocol Validation System.</em></td>
</tr>
<tr>
<td>10h30</td>
<td><strong>COFFEE</strong></td>
</tr>
<tr>
<td>11h00</td>
<td>P.S. Kritzinger, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><em>Protocol Performance using Image Protocols.</em></td>
</tr>
<tr>
<td>11h35</td>
<td>Invited Lecture by Professor G. Lasker, University of Windsor, Ontario.</td>
</tr>
<tr>
<td></td>
<td><em>The Relationship of Natural and Artificial Intelligence.</em></td>
</tr>
<tr>
<td>12h30</td>
<td><strong>LUNCH</strong></td>
</tr>
<tr>
<td>14h00</td>
<td><strong>Artificial Intelligence.</strong></td>
</tr>
<tr>
<td></td>
<td>Chairman: G. Lasker.</td>
</tr>
<tr>
<td></td>
<td>A. Cooper, CSIR</td>
</tr>
<tr>
<td></td>
<td><em>A Data Structure for Exchanging Geographic Information.</em></td>
</tr>
<tr>
<td>14h35</td>
<td>A. I. Newcombe, University of Cape Town and A. Rado, National Library</td>
</tr>
<tr>
<td></td>
<td>of Medicine, Maryland.</td>
</tr>
<tr>
<td></td>
<td><em>Strategies for Automatic Indexing and Thesaurus Building.</em></td>
</tr>
<tr>
<td>15h15</td>
<td><strong>COFFEE</strong></td>
</tr>
<tr>
<td>15h30</td>
<td><strong>Database Systems II.</strong></td>
</tr>
<tr>
<td></td>
<td>Chairman: C. Bornman.</td>
</tr>
<tr>
<td></td>
<td>M.J. Philips, CSIR</td>
</tr>
<tr>
<td></td>
<td><em>An Approach to Direct End-user Usage of Multiple Databases.</em></td>
</tr>
<tr>
<td>16h05</td>
<td>S. Berman, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><em>A Semantic Data Model Approach to Logical Data Independence.</em></td>
</tr>
<tr>
<td>16h45</td>
<td>Open Forum with professors G. Lasker, D. Teichrow and N. Wirth.</td>
</tr>
<tr>
<td></td>
<td>Moderator: Dr. D. Jacobson.</td>
</tr>
<tr>
<td>19h30</td>
<td>Symposium Banquet in Cullinan Room.</td>
</tr>
<tr>
<td></td>
<td>Guest speaker, Dr. D. Jacobson., Group Executive: Technology, Allied</td>
</tr>
<tr>
<td></td>
<td>Technologies Limited.</td>
</tr>
</tbody>
</table>

*Information Systems.*

Chairman: D. Teichrow.

L. du Plessis and C. Bornman, UNISA.

*The ELSIM Language: an FSM-based Language for the ELSIM SEE.*

J. Mende, University of the Witwatersrand.

*Three Packaging Rules for Information System Design.*

*Computer Languages III.*

Chairman: N. Wirth.

W. van Biljon, CSIR.

*Experience with a Pattern-matching Code Generator.*

C. Mueller, University of the Witwatersrand.

*Set-oriented Functional Style of Programming.*
08h00  Registration (Tutorial only).

08h30  Tutorial.
The Tutorial will be given by professor Niklaus Wirth, Division of Computer Science, Swiss Federal Institute of Technology, Zurich.

The use of Modula-2 in Software Engineering.
Topics to be covered include:

What is Software Engineering?
Data types and structures.
Modularization and information hiding.
Definition and implementation parts.
Separate compilation with type checking.
Facilities to express concurrency.
Pompous programming style.
What could be excluded?

12h15  Close of Symposium.

12h30  LUNCH
A GENERALIZED EXPRESSION STRUCTURE

by

W R VAN BILJON
National Research Institute for Mathematical Sciences, CSIR

ABSTRACT
The expression structure of programming languages is examined. Two weaknesses are identified. The first pertains to the inability, or limited ability, to redefine operators or to define new operators. The second pertains to the inconsistency of the levels of abstraction allowed for left values and right values in assignments. The expression structure of backchat, a language for user interface implementation, is introduced. It is shown how this structure alleviates the problems of other programming languages.
1. **INTRODUCTION**

A generalized, extensible expression structure that allows the programmer to define his own level of abstraction is presented. This expression structure is described against the background of the strengths and weaknesses of expressions in current programming languages, and has been incorporated into backchat, a language for user interface design.

2. **EXPRESSIONS IN PROGRAMMING LANGUAGES**

Programming languages are formal notations for problem solving, with the beneficial side-effect that computers can be made to understand the notation, and thus perform the actual solution. The notation plays a part in shaping the thoughts of the person using the language, and it would thus be advantageous to choose a notation that most closely mimicks the abstraction chosen by the user. This section examines expressions of programming languages by focusing on exactly this point. We shall ask: How closely does an expression in a current programming language resemble the problem-related abstraction?

2.1 **ALGOL-LIKE LANGUAGES**

The expression syntax of Algol-like languages is well known and is largely unchanged from FORTRAN. The strong arithmetic heritage of this notation is somewhat restrictive in the more general problem-solving environment. The most obvious of these limitations is the higher status given to operators as opposed to user-defined functions. This prevents users from redefining operators such as '+' to be meaningful in their problem domain. Examples of such redefinition might be in order to perform modulo arithmetic or matrix addition. The inability to provide for the latter example also leads to what Backus termed 'word-at-a-time' programming in his 1977 Turing lecture [1].

Ada [2] provides a partial solution to this by allowing operators to be overloaded by user-defined functions, but this is not entirely satisfactory since the number of parameters of an operator must remain identical. A definition of modulo addition as a ternary operator that will allow

\[ 4 + (5, 6) \]

4 + (5, 6)

to mean 4 plus 5 modulo 6 is not possible. (Note that the meaning of this is different from \((4 + 5) \mod 6\), due to overflow conditions).

Another limitation of Ada's operator overloading is the enforcement of a Boolean result to any relational operator. This disallows, for example, the implementation of tri-state logic whilst maintaining an intuitive notation.

Finally, Ada does not allow the definition of new operators, for example, a post-fix factorial ('!').
The second, and possibly more important, limitation of expressions in Algol-like languages is the lack of uniformity between value naming and variable naming, with the corresponding impact on data abstraction. Consider variables in such a language. One can name a value (right-hand value) contained in a variable by writing the name of the variable, for example:

\[
\begin{align*}
& a + 4 \\
& 2 - a[i] \\
& \text{or} \\
& 2 * a.b;
\end{align*}
\]

Similarly, one can name the variable (left-hand value), for example:

\[
\begin{align*}
& a := 4 \\
& a[i] := 5 \\
& \text{or} \\
& a.b := 3.
\end{align*}
\]

An alternative interpretation (as in Forth) has a variable always naming its left-hand value (or address). This forces a dereferencing operator to be used in all cases in which a value is meant:

\[
a := b\uparrow;
\]

This blemish hampers our use of expressions, and is not the required, or usual, solution.

Now consider a dictionary data structure (possibly implemented by a binary tree or hash table) with a corresponding function \textit{search} that takes a key as parameter and returns the value of the data associated with the key. In most Algol-like languages, we can write:

\[
x := \text{search}(k);
\]

In other words, the language allows a programmer to define an arbitrarily complex data structure and to name values in it. In Pascal, Ada, Modula-2 and most other Algol-like languages the same abstraction mechanism cannot name variables (left values). One cannot write:

\[
\text{search}(k) := x;
\]

The reason is that \textit{search} returns a value (right-hand value) not an address (left-hand value). But, even if we define \textit{search} to return a pointer, we still cannot do the assignment directly. An intermediate pointer must be introduced:

\[
p := \text{search}(k); \\
p\uparrow := x;
\]
In addition, we introduce the same blemish noted above since we must now write a dereferencing operator when naming both values or variables (that is, both right-hand and left-hand values).

This inconsistency is more than a mere notational bother. Bearing in mind that notation (or language) shapes thought, it is clear that the above prevents the user from thinking in his own chosen abstraction. Intermediate values force him back to the abstraction chosen by the notation. Anyone who believes that the two notations above differ only in detail (or are different ways of saying the same thing) must surely believe that Shakespeare's Romeo and Juliet and a Mills & Boon romance are just different ways of writing about the same thing.

2.2 **NON-ALGOL-LIKE LANGUAGES**

From this large class of programming languages, the expression structure of three well-known languages - Forth, APL and Smalltalk - is examined.

Forth is probably best known for its Reverse Polish Notation (RPN). This immediately exposes its major flaw - its reliance on a stack structure and the resulting use of RPN. To be fair, Forth does allow the definition of any new function, and these functions may be declared as immediate (that is, their run-time can be defined as the program's compile time). This feature allows a user to program functions that modify the syntax of Forth expressions by "taking over" the parsing function of the compiler for a period of time. Although this is workable and, in the case of control structures, elegant, it is hardly the most practical of solutions for operators. In addition, Forth suffers from the incessant left-value naming problem mentioned above.

APL provides an extremely rich set of operators and allows the definition of new operators. The notation of an operator in APL is determined according to the valence (or number of parameters) of an operator. If it is unary, it is prefix; if it is binary, it is infix; and higher-order operators are written in the usual way (for example f(n,b,c)).

APL partially solves a number of the above problems, but it still does not allow programmers to design their own notation. It also suffers from the left value/right value inconsistency noted above.

Smalltalk is an object-oriented programming language, incorporating the class concept of Simula to provide its abstraction mechanism. All expressions in Smalltalk are structured as message-passing constructs. An arithmetic expression such as

\[ 2 \times 4 \]
is interpreted as "send the message '*4' to the integer object 2."
By careful construction, it is possible to construct readable
user-defined abstractions, especially since all objects and
messages are regarded as being on an equal footing. The object
obsession of Smalltalk hampers thought in the user's own problem
domain - after all, one seldom thinks of expressions as the passing
around of messages.

3. **EXPRESSIONS IN backchat**

The expression structure of backchat was designed to solve the above
problems by maintaining a structure that is simple to implement and that
maintains a storage model of computation [1] for the effective
manipulation of devices. This was achieved by using three basic

(1) A lexically flexible notation for operators.
(2) User definable operators (or functions) with user specification of
their notation (that is, prefix, infix or postfix).
(3) Operator overloading on the number of parameters, their types, the
operator (or function) result type, the handedness (left or right)
of the parameters, the time of availability of the parameter values
(compile versus run time), and the result.

These three techniques are discussed in the following sections; it is
shown how each one aids users in defining notations that suit their own
abstraction. In the following, the terms function and operator are used
interchangeably.

3.1 **LEXICALLY FLEXIBLE OPERATORS**

Function identifiers in backchat may consist of either a sequence
of alphanumeric characters, starting with an alphabetic character,
or a sequence of special characters. Special characters are chosen
from the set

{#, &,-, +, /, :, <, >, . , = ! $ , ? , @, ”, ' , }.

On a normal ASCII keyboard, this leaves the set

{blank, (,), [,.], ..., comma, :, left brace, right brace, ", end of
line}

for variables and delimiters.

This convention has the result that delimiters must be inserted
between adjacent special character functions. This is not a
problem, since readability usually dictates the insertion of
brackets in such cases, for example:

4 + (-2).

Alphanumeric identifiers and special character identifiers are
separated to obviate the need for a delimiter (such as a space)
between operands and operator, a most annoying necessity in Forth.
Variable identifiers in backchat are always alphanumeric, and the usual notation is used for record and array specification.

3.2 USER DEFINABLE OPERATORS

backchat is a dictionary-based language. In other words, the table of symbols defined in the language is available at development, compile and run time. The structure of the dictionary is in the form of a network of sub-dictionaries, each forming a module of code. Upward arcs in the network are 'depends-on' or 'uses-definitions-of' relations. The topmost dictionary thus contains the pre-defined system primitives, whereas lower dictionaries might contain I/O libraries, etc. Figure 1 illustrates this structure.

![Diagram of dictionary structure]

This structure is a generalization of Forth's dictionary and the Pascal scope structure. Dictionary searches are performed from a current dictionary (which is nominated by the user and which would be the current development environment) upwards toward the system dictionary.

This arrangement allows users to redefine any of the system primitives, or to define functions of their own. A function definition has five essential elements:

(a) a function name;
(b) function parameters, their types and the result type;
(c) the form of the function (infix, prefix, or postfix);
(d) a function body;
(e) execution time directive (compile-time execution or run-time execution.)

For example, a function specification for the integer operator '+' might look as follows:

FUNCTION (integer a) + (integer b) RETURNS (integer):
BEGIN
   
END;
Here '+' is specified as an infix operator with two parameters that has an integer result. The ternary modulo-addition function given as an example earlier can be defined by the heading:

FUNCTION (integer a) + (integer b, m) RETURNS (integer);

Appendix A provides a grammar (in YACC [3] notation) describing an expression written using variables, literals and functions of the above type. It also provides for literal aggregates such as (1,2,3), which would allow vector and matrix operations to be specified in a natural way. The heading of a function that multiplies two three-element vectors may be specified as:

FUNCTION (integer x1,x2,x3) * (integer y1,y2,y3) RETURNS (integer.integer.integer);

The expression syntax has four interesting traits. Firstly, it makes no distinction between operator precedences. This forces the use of brackets when expressions are specified, but it is felt that this is a small price to pay for flexibility. The inclusion of a precedence with each function declaration could lead to unpredictable semantics of expressions. Secondly, the grammar is ambiguous. These exist because the parser cannot distinguish between operators or variables. The problem is solved by choosing an interpretation when the ambiguity arises.

The third aspect of interest is that the expression syntax, together with compile-time functions, is a powerful enough structure to encompass statements. For example, an IF-statement may be defined as a resultless immediate function taking a Boolean parameter. The function would be called by the compiler after the expression's parse tree had been constructed and code had been generated for the parameters of the IF function. It would then, by means of system primitives and a global stack-variable, generate a conditional branch and a backpatch address. A further parameterless function ENDI may then backpatch the branch address. This arrangement is illustrated below:

COMPILER FUNCTION IF (LAZY (boolean) condition);
  BEGIN --("-- is a comment compiler function that ignores its parameter");
    gen_code (branch_false, condition, 0,);--("generate the branch");
    if _back_patch_stack < (pc-4);--("<| pushes a value");
  ENDF;

COMPUTER FUNCTION ENDI;
  BEGIN
    back_patch (if_back_patch_stack |);--("| pops the stack");
  ENDF;
The "LAZY" specification in the heading indicates that the value of the parameters will only be available at run-time even though the function executes at compile time.

The final point of interest is that no distinction is made between a left-hand side and a right-hand side. This implies that, since statements are incorporated into the expression structure, the assignment operator is regarded as just another function. This leads us to the consideration of overloading, the mechanism used in backchat for left value/right value resolution.

3.3. FUNCTION OVERLOADING

Overloading has already been mentioned in the context of Ada. Apart from its use as a mechanism to make a notation more usable (by allowing the name '+' to denote both integer and real addition, for example), it can also be used to resolve the distinction between the naming of values and variables. The following sections describe this use overloading by sketching its development.

The first solution stated the overloading rule as follows:

A function is resolved on the basis of its name, the and order number of parameters and their types, and the result type. In addition, a variable is defined as a parameterless function with two definitions:

1. It returns a value of the type of the variable.
2. It returns a pointer to a value of the type of the variable (the variable's address).

At first glance these two rules seem to solve the problem. An expression such as

\[ a := b \]

where \( a \) and \( b \) are integers, is resolved by invoking the function defined as:

FUNCTION (pointer_to_integer a) := (integer b);  
BEGIN  
  Perform move of integer to address pointed to by a;  
ENDF;

and using the first function associated with the variables above for \( b \), and the second for \( a \). Another interesting situation is created by the invocation of the dereferencing operator. Using the post-fix '↑' notation of Pascal, if we write:

\[ a↑ := b↑; \]

where \( a \) and \( b \) are pointers to integers, then the '↑' on the left has a different semantics (in fact, it is the null operator) to the '↑' on the right, which performs an actual dereference. This situation is resolved by the ':=' noted previously and the following two definitions of '↑':
FUNCTION (pointer_to_integer a) \rightarrow \text{RETURNS} \text{pointer_to_integer};
BEGIN
 \text{RETURN}(a);
ENDF;

FUNCTION (pointer_to_integer a) \rightarrow \text{RETURNS} \text{integer};
BEGIN
 \text{RETURN} (\text{dereferenced value of a})
ENDF;

Unfortunately, a more complex arrangement with a double dereference runs into trouble:

\( (a\uparrow)\uparrow := (b\uparrow)\uparrow; \)

is ambiguous if another ':=' function for assignment of pointers to integers is available since a valid resolution could include one invocation of the first ':=' function above on the right-hand side of the expression. If we identify the dereferencing operators above by \( \uparrow_1 \) and \( \uparrow_2 \) respectively, such a resolution could be

\( (a\uparrow_1)\uparrow_1 := (b\uparrow_2)\uparrow_1. \)

This is clearly not the required semantics.

The solution to this problem is to include a specification of the \textbf{handedness} of a function in its parameter definition and its result definition. In the function headings that follow, we prefix a left-hand value by \textbf{LHV}. Any parameter or function result is assumed to be a right-hand value, unless the prefix \textbf{LHV} is present. A left-hand value is defined to be an address of a value, whereas a right-hand value is the value itself. This definition is similar to the \textbf{VAR} (or call-by-reference) specification of parameters in Pascal, or the \textbf{OUT} (or \textbf{IN OUT}) specification of parameters in Ada (although the implementation need not be call-by-reference). In backchat this concept is extended to function results.

The rule for overloading of functions now becomes:

A function is resolved on the basis of its name, the number and order of parameters and their types and handedness, and the result type and its handedness.

It is interesting to note that in the definition of overload resolution in Preliminary Ada [4] the mode (\textbf{IN}, \textbf{OUT} or \textbf{IN OUT}) also had to be taken into account. This has been dropped in the Ada reference manual.

A variable is still regarded as two functions:

(1) It returns a value of the type of the variable.
(2) It returns a left-hand value (\textbf{LHV}) of the type of the variable.
This overloaded definition of variables is well known and used in
Pascal since the semantics of a variable as a VAR-parameter differs
from one as a value parameter.

We can now define the assignment and dereferencing functions above as
follows:

\[
\text{FUNCTION } (\text{LHV (integer) a} := (\text{integer b}) ;
\text{BEGIN}
    \text{Perform move of integer;}
\text{ENDF ;}
\]

\[
\text{FUNCTION } (\text{LHV (pointer_to_integer) a} := (\text{pointer_to_integer b}) ;
\text{BEGIN}
    \text{Perform move of pointer;}
\text{ENDF ;}
\]

\[
\text{FUNCTION } (\text{pointer_to_pointer_to_integer a}) \uparrow \text{RETURNS (pointer_to_integer)} ;
\text{BEGIN}
    \text{RETURN (dereferenced value of a);}
\text{ENDF ;}
\]

\[
\text{FUNCTION } (\text{pointer_to_integer a}) \uparrow \text{RETURNS (integer)} ;
\text{BEGIN}
    \text{RETURN (dereferenced value of a);}
\text{ENDF ;}
\]

\[
\text{FUNCTION } (\text{pointer_to_integer a}) \uparrow \text{RETURNS (LHV (integer)) ;}
\text{BEGIN}
    \text{RETURN (LHV(a)); \text{-- ("LHV performs a type conversion from a pointer to a left-value") ;}
\text{ENDF ;}
\]

As before we identify these functions by \( :=_1, :=_2, \uparrow_1, \uparrow_2 \), and \( \uparrow_3 \),
respectively. If a and b are of type pointer_to_integer, then the
assignments are resolved as

\[
a :=_2 b;
\]

\[
a\uparrow_3 :=_1 b\uparrow_2
\]

If a and b are of type pointer_to_pointer_to_integer, we have

\[
(a\uparrow_2) \uparrow_1 :=_1 (b\uparrow_1)\uparrow_1
\]

and

\[
a\uparrow := b\uparrow
\]

is unresolvable, since we have not defined a \( \uparrow \) that returns (LHV
pointer_to_integer).
The mechanism works because LHV provides an escape from the normal 'pointer_to' type structure, whilst maintaining pointer semantics. It is therefore also powerful enough to be used when a type compatibility scheme that finds all pointer types compatible is employed.

This mechanism also allows, as is demonstrated by the definitions of 'T', any level of abstraction on the left-hand side. The search example mentioned earlier may be implemented by simply declaring search as:

```
FUNCTION search (key k) RETURNS (LHV data);
```

The above examples show one danger in this definition - the unbounded proliferation of function definitions. This is solved by employing a polymorphic type structure - an aspect which is not described in this paper.

4. EVALUATION AND CONCLUSION

Hoare [5] suggests six criteria for judging the structure of expressions:

(1) transparency of meaning;
(2) transparency of purpose;
(3) independence of parts;
(4) recursive application;
(5) narrow interfaces;
(6) manifestness of structure.

Points 4 and 6 are evident from the grammar in Appendix A, at least as far as structure is deemed necessary. Transparency of meaning and purpose is guaranteed by allowing expressions at the problem's level of abstraction. Independence of parts is achieved, with the exception of overloading of functions where context sensitive constructs may be resolved. The added expressiveness and abstraction power gained by this is judged to outweigh the cost by far. Finally, narrow interfaces are provided since the interface merely consists of the type of a value (or values), plus a possible specification of its handedness. This is no wider than the function and operator interfaces of Pascal or Ada.

As far as compiler considerations are concerned, the grammar is resolvable if ambiguities are resolved by pre-defined choice. The resolution of the overloading of functions is thus a post-parsing operation, and is no more complex than the resolution of overloading in Ada, a problem that has been solved elegantly.

A generalized structure for expressions in backchat, a language for user interface production, has been presented. The structure solves all difficulties arising from the intuitive context-sensitivity of assignment statements and allows expressions that exhibit an arbitrarily high level of abstraction. This structure is appropriate for a language
for user interface production since screen, icon and window manipulation require expressions with a high level of abstraction (a level that may change with changes in hardware) whilst maintaining a storage model of computation to manipulate bitmaps effectively.
APPENDIX A

Following is the BNF of backchat expressions in YACC format. A colon denotes the usual '::=' and a semicolon the end of a production.

%token ALPHAID SPECIALID VALUE
%token LBRAC RBRAC LSQRAC RSQRAC PERIOD COMMA SEMICOL
%type infix prefix postfix exprlist factor operator expression variable
%start expression

expression  :  infix
            |  prefix
            |  postfix
            |  factor
            ;
postfix   :  factor operator
            ;
prefix    :  operator factor
            ;
infix     :  factor operator expression
            ;
factor    :  LBRAC EXPRLIST RBRAC
            |  variable
            |  VALUE
            ;
operator  :  SPECIALID
            |  ALPHAID
            ;
exprlist  :  exprlist COMMA expression
            |  expression
            ;
variable  :  ALPHAID
            |  variable PERIOD ALPHAID
            |  variable LSQRAC expression RSQRAC
            ;
REFERENCES


