Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels
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, TeX or LaTeX 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, TeX or LaTeX 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.
Regardless of the size of your computer system it is a strategic tool. We are committed to helping you use it most effectively. All it needs is a little imagination.

UNIDATA
Telephone (012) 45-1152

THERE'S A GIANT IN EVERY ONE OF US
The Department of Computer Science at Pretoria University is inviting applications for two annually awarded bursaries for MSc study. These bursaries are aimed at outstanding students who have a Computer Science Honours degree (or equivalent qualification). As far as possible, one such bursary will be awarded to a South African student, while the other will be awarded to a student from another African state. The hope is to thereby promote interaction between South African computer scientists and computer scientists elsewhere in Africa.

The bursaries are funded by the Department’s contract research work and will amount to a minimum of R20000 per year. They are intended to cover costs for tuition, housing, transport, books, and an adequate allowance for other living expenses. Successful candidates will be required to work for 20 hours per week as part of the Computer Science Department’s contract research team. The work assigned to each candidate will be aimed at providing him with practical experience in his field of interest.

Study fields for which bursaries will be considered include:
- Computer Networking
- Distributed Systems
- Software Engineering
- Artificial Intelligence
- Intelligent Computer Aided Instruction
- Machine Aided Translation

Students should aim to complete the dissertation that is required for an MSc within two years of study (i.e. one year of full time study). The dissertation should preferably be written in English but it may also be written in Afrikaans.

The continuation of the bursary and adjustments to the allowance will be decided upon annually and will depend on progress. It is a condition of the bursary that, after completing his/her studies, the student should return to his/her country of origin for the same number of years for which the bursary was held.

Applicants may request application forms from:

MSc. Bursaries
The Computer Science Department
University of Pretoria
Hillcrest
Pretoria
0083
South Africa
Editor's Notes

It is with sincere gratitude that SACJ takes leave of Dr Peter Lay who, until recently, was the assistant editor dealing with Information Systems. He has left academia for what sounds like a more gentle lifestyle. (He has gone farming!) Under Peter's stewardship the number of high-quality IS papers in SACJ grew steadily. In general, IS papers tend to be accessible and relevant to a wide spectrum of computer professionals, and the quality of IS papers that have been appearing in SACJ has significantly contributed to the increased interest being shown in the journal by the local computer industry. If this growth in interest is to be sustained, it is urgent and important to find a suitable replacement assistant editor. The ideal candidate should not only be respected as an academic by his peers, but should also be disposed to enthusiastically promote SACJ in the private sector. Since a shortlist of candidates is currently being compiled, I would like issue a general appeal for names that might be included on it. Please contact me urgently if you would like to be considered for the job, or if you would like to nominate someone that you consider to be particularly suitable.

My three year term of office as editor expires in October. I have always considered it a great privilege to hold this position, and as a result, I felt honoured when the SAICS executive committee requested that I stay on for a further term. Nevertheless, I initially declined the request on the grounds that the time-demands of the job were significantly eroding my ability to fulfil other duties. Particularly demanding has been the task of seeing to the typesetting of the various contributions - either by doing it myself, or by ensuring that it is adequately done by someone else. Recently, however, Prof G de V Smit (Riel Smit) at UCT has offered to assume the role of production editor. This generous offer so much changes the complexion of what is being asked of me that I am now both willing and honoured to continue as editor for another term. I am very grateful to Riel for his offer and I look forward to working with him. In future, authors whose papers have been accepted for publication will be asked to liaise directly with him regarding the precise form in which the final contribution should be submitted.

The next issue of SACJ will consist largely of a selection of papers that were presented at the 6th South African Computer symposium. The selection will be based on comments from the referees who, at the time, were asked to adjudicate the papers in terms of their appropriateness for both the conference as well as for SACJ publication. Papers which, in the opinion of one or more referees, required major revision will have to be resubmitted to SACJ for refereeing purposes. Authors will soon be contact in this regard.

At the time of writing, the updated list of "approved" publications for the first half of 1991 had not yet been released by the relevant authorities. For the sake of past, present and future contributors I sincerely hope that SACJ will be on the list when it eventually comes out. However, I have become increasingly aware that there is a real danger of laying too much store on papers published in so-called approved journals as a basis for evaluating and rewarding research. I hope to expand more fully on this theme in a future edition of SACJ. Keep watching this space!

Derrick Kourie
Editor

This SACJ issue is sponsored by
a generous donation from
UNIDATA

SACJ/SART, No 5, 1991
The Placement of Subprograms by an Automatic Programming System

J. P. du Plessis and H. J. Messerschmidt
Department of Computer Science, University of the OFS, Box 339 Bloemfontein 9300

Abstract

Programming in a procedural language such as Pascal usually involves the development of subprograms. An automatic programming system needs a systematic method to determine the placement (position) of a subprogram in a program. Such a systematic placement method is described and an example is given how EXPROG, a prototype automatic programming system, uses it.

Keywords: automatic programming, placement of subprograms, program derivation, programming system

Computing Review Categories: D.1.2, I.2.2

Received August 1990, Accepted June 1991

1 Introduction

A program specification gives the requirements that a program has to meet. In writing a program, these specifications are analyzed to determine what the program has to do. The results obtained from this analysis are then used to synthesize a program, i.e. to derive the necessary program statements that solve the problem and thereby satisfy the specification [4].

This article is concerned with how to synthesize a program by means of a computer program. A computer program that is used to synthesize another program is called an automatic programming (AP) system [1]. In this article we shall therefore discuss the automatic synthesis of programs by means of an AP system.

The development of a Pascal program usually involves the creation of subprograms. The placement of these subprograms in the source code is important, because a subprogram needs to be defined before it can be referenced (called) [3]. A human programmer uses the scope rules of Pascal to determine the position (placement) of the subprograms in the Pascal program. An AP system therefore needs a systematic way to position the subprograms according these scope rules. While the placement of subprograms is easily solved by a human programmer, this paper will show that it is not trivial for an AP system.

Another strategy, inline code expansion, was not considered for the following reasons:

- It would result in a large, space-inefficient program.
- The principle of modular programming would be violated.
- The overhead costs of generating appropriate variables, etc. for subprogram invocation are not reduced by the use of this strategy.

In this article we shall give a brief description of how EXPROG ([2]), a prototype AP system, works. We shall then describe and illustrate by example how the problem of placement of a subprogram is solved in EXPROG.

2 How EXPROG works

EXPROG ([2]) is a prototype AP system that writes Pascal programs. Its main activity, however, is to develop an algorithm and then convert it into a Pascal program. Such an algorithm consists of a main program together with a number of subprograms (subalgorithms).

EXPROG has access to general programming knowledge, as well as domain knowledge - stored in its knowledge base. EXPROG constructs programs for elementary scientific problems, e.g. computing an average, a standard deviation, implementing scientific formulas, etc. EXPROG's domain knowledge is stored in the form of EXPROG definitions ([2]). This domain knowledge is extracted from the domain expert and then transferred to EXPROG's knowledge base via the knowledge engineer. In these EXPROG definitions, the domain expert describes how a problem can be solved, namely by describing the necessary algorithmic constructs to be used. In these constructs the domain expert may refer to other EXPROG definitions or use predefined generic subprograms (to be discussed in the next section.)

The specification of the Pascal program is given to EXPROG via a user interface. The user interface recognizes only a small subset of English words, and only simple type sentences are allowed. For example, to specify a Pascal program to determine and print the average and sum of a list of N real numbers, the following specification could be given to EXPROG:

Consider a list of N real numbers.
Determine the average.
Print the sum.

Having read and interpreted the program specification, EXPROG starts the analysis phase. During this phase the problem is analysed by using the general programming knowledge and the domain knowledge [2].

The next phase is the synthesis phase. During this phase the target algorithm is formed, representing the main program and the subprograms of the final Pascal program. The order in which the subprograms appear in the algorithm is the same as that in which they should appear in the Pascal program. This is made possible by the construction of a subprogram tree (which will be described later) during the synthesis phase. The last step in the synthesis phase is the conversion of the target algorithm into the final Pascal program.

### 2.1 Subprogram Instances

EXPROG has access to a library of predefined generic subprograms. These subprograms are constructed by the knowledge engineer, assisted by the domain expert.

By allowing generic subprograms, EXPROG is able to refer to subprograms having the same name, but different formal parameter declarations, e.g.

EXPROG may refer to two functions by the name of MAXIMUM - one dealing with integer numbers, the other dealing with real numbers.

The heading of an example generic subprogram, PP1, with formal parameters FP1, FP2 and FP3 is:

```pascal
PROCEDURE PP1 (IN FP1 Real; IN FP2 : Integer; IN-OUT FP3 : AR1);
```

In this heading the subprogram is identified as a procedure subprogram by the word PROCEDURE. The only other alternative is FUNCTION for a function subprogram. Next to the name of the subprogram appears the list of formal parameters, separated by semi-colons. Each formal parameter description consists of three fields:

1. The words IN, IN-OUT or OUT identifying a formal parameter as being called by value (IN) or called by reference (IN-OUT or OUT).
2. The name of the formal parameter, e.g. FP1.
3. The data type and data structure expected of the actual parameter. The word SCALAR means that the actual parameter expected should be a scalar, e.g. a REAL or an INTEGER. If the data type is specified, e.g. INTEGER, an integer value or integer variable is expected as the actual parameter, depending on the words IN, IN-OUT or OUT. If the word ARRAY appears it means that an array of scalar elements is expected.

As an example consider the generic subprogram PP1 as described above. Assume that PP1 is called with actual parameters as follows:

Call PP1 (A,B,C).

If the variables A, B and C are declared as follows:

- A : Real;
- B : Integer;
- C : Array[1..20] of Real

then EXPROG creates a subprogram instance, called PP1, having a heading with formal parameters declared as follows:

```pascal
TYPE AR1 = Array[1..20] of Real;
PROCEDURE PP1 (IN FP1 : Real; IN FP2 : Integer; IN-OUT FP3 : AR1);
```

A generic subprogram consists of the following parts: a name, a formal parameter list, a list of local variables the declarations of which depend on the formal parameters, a list of local variables the declarations of which do not depend on the formal parameters, and a body part that consists of a number of algorithmic statements.

The creation of a subprogram instance from a generic subprogram is formally described as the process of creating a subprogram with the same name as the generic subprogram. It also involves the matching of the actual parameter declarations with the formal parameter descriptions of the generic subprogram to form the formal parameter list of the subprogram instance. In the case of a successful match these formal parameters inherit the declarations of the actual parameters. If the match is unsuccessful an error will occur, e.g. if the call to PP1 in the previous example was made as Call PP1(B,A,C) then an error would occur during the matching of the actual parameter A (a real scalar) and the formal parameter FP2 (expecting an integer scalar). In such a case no subprogram instance is created.

Because the declarations of some of the local variables may depend on the declarations of the formal parameters, these local variables may differ from one instance to another. Apart from these differences, the rest of the subprogram is the same for each instance. Pascal cannot handle more than one subprogram with the same name but different formal parameter declarations.

When EXPROG creates a subprogram instance from a generic subprogram, it also records, in a list, the name of the subprogram instance, the formal parameters and the local variables as declared for the subprogram instance. This list is called the subprogram list. It also records a reference to the subprogram (or main program) from where the call to this subprogram instance originates. This subprogram list is used during the synthesis phase to construct a subprogram tree. EXPROG then scans the body of this subprogram instance for any call it makes to other generic subprograms.

For each of these calls encountered, EXPROG creates a subprogram instance in the same way as described in the previous paragraph. This process is
Figure 1: An example of a subprogram tree.

repeated until all the subprogram instances are created and recorded on the subprogram list. Because of this process, identical subprogram instances, having the same name and identical formal parameter declarations, may occur. Note that these identical subprogram instances may fall within different scopes, i.e., they may be referenced by different subprograms. Eventually, however, EXPROG ensures that the unique subprogram instance of a group is placed in the subprogram tree in such a way that it falls within the scope of each subprogram that calls it.

The eventual position of the unique subprogram instance in a subprogram tree is determined by the theorem presented below. Before it is possible to discuss this theorem, a number of definitions need to be given.

During the development of the algorithm, EXPROG encounters calls made to the generic subprograms in its knowledge base. For each such call EXPROG creates a subprogram instance. As mentioned before, there could be several instances of the same subprogram that are identical, but they may fall within different scopes. EXPROG keeps track of all these instances by means of the subprogram list.

Definition 1: A Subprogram Tree is a tree of which the root (MAIN) represents the main Pascal program in the target algorithm, while its nodes represent the different subprograms of the final Pascal program of the target algorithm. Suppose S and J are nodes in a subprogram tree representing two different subprograms. S is in the scope of J if and only if one of the following holds:
- S is a child of J;
- S is a left brother of J;
- S is a left brother of some ancestor of J.

The scope of subprogram J is the set of subprograms that can be referenced by J.

Definition 2: The Path to a Subprogram describes the path, in the subprogram tree, that must be followed to locate the node that represents the subprogram in the tree, beginning at the root. The path is a list of ancestors in the tree of the subprogram concerned. The oldest ancestor, always the root, appears first in the path. For example, in Figure 1 the path to the subprogram P22 is described as MAIN-P2.

Definition 3: The Level of a Subprogram in the subprogram tree is the depth of the node, representing the subprogram, in the tree. The level of the root is 0. The level of the nodes forming the children of the root is 1.

Definition 4: The Length of a Path is the number of ancestors in that path.

During the development of an algorithm, EXPROG records the detail of every subprogram instance in the subprogram list. By using the information in
P2X P2Y

Figure 2: The subprogram tree after the insertion of P222.

the subprogram list, EXPROG is able to construct an initial path for each of the subprogram instances. An initial path consists of a list of ancestor subprograms, the oldest one being MAIN and the youngest one being the subprogram that calls this subprogram for which the initial path is constructed. This list of subprograms may contain identical subprogram instances that may fall within different scopes.

While constructing the subprogram tree, EXPROG performs two types of operations on the tree, viz. an insertion and a deletion operation. To move a subprogram already in the tree from its current position to a new position in the tree involves a deletion of the subprogram at its current position and then an insertion of the subprogram at its new position.

Insertion: Inserting a subprogram in the subprogram tree. When a subprogram S is to be inserted in the subprogram tree, the path of S (specifying also the level of insertion) must be provided. S is inserted, together with its subtree (if any), at that level, to the left of all the other subprograms that have the same parent as S (and are therefore at the same level as S). If the node representing S is the root of a subtree, those subprograms in the subtree will now automatically be in the tree.

To illustrate the insertion operation, refer to Figure 1. Suppose the subprogram P22 calls the subprogram P222 (with its path described as MAIN-P2-P22) and that P222 calls two other subprograms P2X and P2Y. Suppose further that P222 is to be inserted in the subprogram tree. The subprogram tree which results from executing the insertion operation is shown in Figure 2.

Deletion: Deleting a subprogram from the subprogram tree. The node representing S is removed from the tree. At the same time, if S is a root of a subtree, that subtree is removed automatically.

As an example of the delete operation, consider again the subprogram tree as depicted in Figure 1. Suppose that the subprogram P11 (having a path described as MAIN-P1) is to be deleted from the subprogram tree. The subprogram tree which results from executing the deletion operation is shown in Figure 3.

To place the subprograms in the subprogram tree, EXPROG starts with an initial tree of only one node, called MAIN, that represents the main program. The youngest ancestor (PARENT) of a subprogram that is to be placed in the tree, will already be in the tree. By using the information of the initial path, EXPROG is able to ensure that this is the case.

EXPROG executes the following procedure to construct the subprogram tree:

{Note: EXPROG has a list of subprogram instances together with the name of the calling subprogram in the subprogram list.}

REPEAT
1. Let S be the next subprogram instance in the subprogram list.
2. Search the subprogram tree, as constructed up to this point, for a subprogram instance J such that S and J are identical instances of the same generic subprogram. (J and S may fall within different scopes.)
3. IF such a subprogram J was found THEN
   1. Determine the new position of J in the subprogram tree using theorem 1.
   2. Delete J from its current position in the subprogram tree.
   3. Insert J in the subprogram tree at the new position as determined in step 3.1.
ELSE
4. Insert S in the subprogram tree as a descendant of its calling subprogram (PARENT) or as a descendant of the root (MAIN) if no calling subprogram is given.

UNTIL the subprogram list is empty.

Theorem 1: Suppose a subprogram instance S is to be inserted in the subprogram tree. Suppose a subprogram instance, J, already exists in the subprogram tree, such that S and J are identical instances of the same generic subprogram. Suppose also that the sub-
program that calls $S$ (name it PARENT) is already in the tree. Let pathS represent the path to the subprogram $S$, should $S$ be inserted in the tree as a child of PARENT. Let AccS be a set of subprograms such that $S$ would be in the scope of each element of AccS were $S$ to be directly inserted at PathS in the subprogram tree. Let pathJ be the current path to the subprogram $J$ in the subprogram tree. Let AccJ denote the set of all the subprograms in the tree that include $J$ in their scopes, i.e. the set of all the subprograms in the subprogram tree that can reference $J$.

Let \( \text{pathJ} = \text{pathS} \cap \text{pathJ} \).

Suppose that the subprogram tree is reorganised by deleting the subtree rooted in $J$ from its current position, and by re-inserting this subtree into the tree so that the new path of $J$ is \( \text{pathJ}_1 \). Then all the subprograms in AccS and AccJ can reference $J$ at its new position.

**Proof:** PARENT is the subprogram that calls subprogram $S$, and therefore is the subprogram that should have $S$ included in its scope, should $S$ be inserted in the tree. PathS is therefore the path to PARENT concatenated with PARENT. PARENT is therefore an element of AccS.

Consider the two cases where \( \text{pathS} = \text{pathJ} \) and where \( \text{pathS} \neq \text{pathJ} \).

a). The first case is where \( \text{pathS} = \text{pathJ} \). This means that $S$ was to be inserted in the same subtree, and therefore has the same parent, PARENT, as $J$. Now \( \text{pathJ}_1 = \text{pathS} \cap \text{pathJ} = \text{pathJ} \).

Inserting $J$ in its new place means that it will be to the left of all its brothers — the same position where $S$ should have been inserted. PARENT can reference subprogram $J$ at its new position in the tree. Subprogram $J$, in its new position \( \text{pathJ}_1 \), can also be referenced by all the subprograms in AccJ that could reference $J$ before it was moved, according to definition 1. Thus the theorem is valid when \( \text{pathS} = \text{pathJ} \).

b). Consider now the case where \( \text{pathS} \neq \text{pathJ} \). Then \( \text{pathJ}_1 = \text{pathS} \cap \text{pathJ} \).

The subprogram tree will now have to be reorganised to reflect the scopes of the subprograms represented in it. Let Y be the last (youngest) ancestor in \( \text{pathJ}_1 \). Y is then the last ancestor which \( \text{pathS} \) and \( \text{pathJ} \) have in common. There will always be a $Y$, because the oldest ancestor of any subprogram is the root, MAIN. The subtree rooted in $J$ is inserted to the left of all the other nodes that form the children of $Y$. All the subprograms in AccJ will still be able to reference $J$ (have $J$ included in their scopes), according to definition 1. According to definition 1 PARENT will also include $J$ in its scope and can, therefore, reference subprogram $J$ at its new position in the tree. This proves the theorem.

### 4 Examples

To illustrate the use of this theorem, consider the following examples which involve the use of the three generic subprograms, $PP1$, $PP2$ and $FUNC1$, with formal parameters $FP_i$. The headings of these three subprograms are:

- **PROCEDURE $PP1$**
  
  \[
  \begin{align*}
  \text{PROCEDURE } & PP1 \ (\text{IN } FP1 : \text{Scalar;} \\
  & \quad \text{IN } FP2 : \text{Scalar;} \\
  & \quad \text{IN-OUT } FP3 : \text{Array})
  \end{align*}
  \]

- **PROCEDURE $PP2$**
  
  \[
  \begin{align*}
  \text{PROCEDURE } & PP2 \ (\text{IN } FP1 : \text{Scalar;} \\
  & \quad \text{IN } FP2 : \text{Integer;} \\
  & \quad \text{OUT } FP3 : \text{Array})
  \end{align*}
  \]

- **FUNCTION $FUNC1$**
  
  \[
  \begin{align*}
  \text{FUNCTION } & FUNC1 \ (FP1 : \text{Array}) : \text{Integer}
  \end{align*}
  \]

Refer to a section 2.1 for a description of the format of a generic subprogram heading.

Assume the variables $A$, $B$ and $C$ are declared in MAIN as follows:

\[
\begin{align*}
\text{TYPE } \text{AR1} & = \text{Array}[1..20] \text{ of Real;} \\
 & \quad \text{A : Real;} \\
 & \quad \text{B : Integer;} \\
 & \quad \text{C : AR1}
\end{align*}
\]

Now suppose that MAIN calls $PP1$ as follows:

\[
\text{Call } PP1 \ (A, B, C).
\]

An instance of the generic subprogram $PP1$ is created. The heading of the subprogram instance $PP1$ will be as follows:

\[
\begin{align*}
\text{PROCEDURE } & PP1 \ (\text{IN } \quad FP1 : \text{Real;} \\
 & \quad \text{IN } \quad FP2 : \text{Integer;} \\
 & \quad \text{IN-OUT } FP3 : \text{AR1})
\end{align*}
\]

Let us now assume MAIN also makes a call to the subprogram $PP2$ as follows:

\[
\text{Call } PP2 \ (A, B, C).
\]

An instance of $PP2$ is created, resulting in the following heading:

\[
\begin{align*}
\text{PROCEDURE } & PP2 \ (\text{IN } \quad FP1 : \text{Real;} \\
 & \quad \text{IN } \quad FP2 : \text{Integer;} \\
 & \quad \text{OUT } FP3 : \text{AR1})
\end{align*}
\]

The path of $PP1$ and $PP2$ is MAIN. EXPROG now examines the subprogram instance $PP1$. Assume further that the following local variables of $PP1$ are declared. (Only those that are relevant are considered.)

\[
\begin{align*}
\text{TYPE } \text{AR2} & = \text{Array}[1..100] \text{ of Integer;} \\
 & \quad \text{FP5 : AR2;} \\
 & \quad \text{FP8 : AR1}
\end{align*}
\]

Assume subprogram $PP1$ calls $PP2$ and $FUNC1$ in the following ways:

\[
\begin{align*}
\text{Call } & PP2 \ (FP1, FUNC1(FP3), FP5) \quad \ast \text{13.6} \\
\text{FP6 := } & FUNC1(FP8)
\end{align*}
\]

The heading of the subprogram instance $PP2$ is now as follows:

\[
\begin{align*}
\text{PROCEDURE } & PP2 \ (\text{IN } \quad FP1 : \text{Real;} \\
 & \quad \text{IN } \quad FP2 : \text{Integer;} \\
 & \quad \text{OUT } FP3 : \text{AR2})
\end{align*}
\]
This instance of PP2 is different from the previously created one. Call this instance PP2' to differentiate between the two instances. The path of PP2' is MAIN-PP1.

Two calls have been made to FUNC1, both having identical formal parameters. Two identical instances of FUNC1 were created:

1. FUNCTION FUNC1 (FP1 : AR1) : Integer
2. FUNCTION FUNC1 (FP1 : AR1) : Integer

Only one instance is therefore needed. Note that PP1 includes both of these subprogram instances in its scope. The two identical instances are then reduced, using theorem 1, to only one subprogram in the subprogram tree. The heading of FUNC1 is as follows:

FUNCTION FUNC1 (FP1 : AR1) : Integer.

The path of FUNC1 is MAIN-PP1. The subprogram tree as constructed up to now is shown in figure 4.

Let us further suppose that the subprogram instance PP2, with its path as MAIN, also issues a call to FUNC1 as follows:

... + FP1 * FUNC1(FP3).

An instance of FUNC1 is therefore created and the heading is as follows:

FUNCTION FUNC1 (FP1 : AR1) : Integer.

Note that this instance of FUNC1 is identical to the first instance of FUNC1 that was created and which is shown as FUNC1 in the subprogram tree of figure 4. This new instance of FUNC1 falls within a different scope than the scope of the first instance of FUNC1.

FUNC1 is already in the subprogram tree, therefore EXPROG uses theorem 1 to determine the new position of FUNC1, making sure that it falls within the scope of both PP1 and PP2. The resulting subprogram tree is shown in figure 5.

As was assumed above of subprogram PP2, subprogram instance PP2', with its path MAIN-PP1, also issues a call to FUNC1 as follows:

... + FP1 * FUNC1(FP3).

This means that another instance of FUNC1 is created, which has the following heading:

FUNCTION FUNC1 (FP1 : AR2) : Integer.

Clearly this instance differs from the previous instance of FUNC1 already in the subprogram tree. To differentiate between these two instances of the generic subprogram FUNC1, name this last instance FUNC1'. The path of FUNC1' is MAIN-PP1-PP2'.

EXPROG inserts FUNC1' in the subprogram tree which results in the completed subprogram tree, shown in figure 6.

The subprograms call each other as follows:

MAIN calls PP1 and PP2.
PP1 calls PP2' and FUNC1.
PP2 calls FUNC1.
PP2' calls FUNC1'.

The scope of each of these subprograms can now be determined from the subprogram tree in figure 6:

Scope of MAIN : PP1, PP2 and FUNC1.
Scope of PP1 : PP2, FUNC1 and PP2'.
Scope of PP2 : FUNC1.
Scope of PP2' : FUNC1', PP2 and FUNC1.

Given this information it is clear that each subprogram, called by another subprogram, is in the scope of that calling subprogram.

It should be noted that when a subprogram is moved to a new position it may no longer able to reference subprograms which it formerly could reference. Although such a situation is possible, it is claimed that the validity of the algorithm on page 4 remains valid.
unaffected.

For a specific example of such a situation, refer to figure 4. Here PP2 is in the scope of FUNC1, whereas in figure 5 FUNC1 is in the scope of PP2. Note, however, that although PP2 is in the scope of FUNC1 in figure 4, FUNC1 does not actually call PP2. Each subprogram that actually calls other subprograms can still reference those subprograms directly.

To further illustrate the continued validity of the algorithm on page 4, consider the following scenario:

Suppose A, B, J and S are subprograms and that A calls B and S, B calls J and J calls S. Figure 7 shows the partial subprogram tree. Notice that S is in the scope of J. Suppose that A calls J as well. The result is a tree having J as a child of A, but to the left of S. S is no longer in the scope of J and J cannot reference S directly anymore.

However, EXPROG first assembles all the subprograms in the subprogram list and then starts construction of the subprogram tree. Since A is the oldest ancestor, its call to J is placed on the subprogram list before calls made by any of its descendants are placed on the subprogram list. This ensures that the call of A to J is dealt with first by EXPROG and it would result in the correct subprogram tree. EXPROG deals with these above-mentioned calls in the correct order: A calls B, S and J. B calls J. J calls S. This results in the correct partial subprogram tree as shown in figure 8. The scope of each subprogram includes the subprogram(s) according to the calls made.

EXPROG therefore produces the correct subprogram tree by using the algorithm on page 4.

5 Summary

This paper describes how the placement of a subprogram in a program can be determined by an automatic programming system. In the construction of the subprogram tree, and in particular the placement of subprograms in that tree, theorem 1 plays an important role by determining the position of the subprogram in the tree, thereby satisfying the requirement that a subprogram must be defined before it can be referenced.

Figure 7: The partial subprogram tree.

Figure 8: The correct subprogram tree.

This theorem has proved useful for EXPROG in the development of an algorithm, determining the correct placement of the different subprograms in the final program. It should, therefore, play an important role in the placement of subprograms during the development of Pascal-like programs by automatic programming systems.

EXPROG does not allow the use of recursive subprograms at this stage — theorem 1 is therefore not valid for recursive subprograms. For theorem 1 to be able to handle recursive subprograms, some special checking mechanisms need to be used in order to determine which subprograms are recursive.

6 Acknowledgements

We wish to acknowledge the valuable advice and suggestions from the two anonymous referees. This work has been supported by a research grant from the University of the OFS.

References

Contents

GUEST CONTRIBUTION

Why all the Fuss About Neural Networks?
G Barth ........................................... 4

RESEARCH ARTICLES

The Placement of Subprograms by an Automatic Programming System
J P du Plessis and H J Messerschmidt .......................................................... 9

An Investigation into the Separation of the Application from its User Interface
J H Greyling and P R Warren .......................................................... 16

The Physical Correlates of Local Minima
L F A Wessels, E Barnard and E van Rooyen .................................................. 22

An Efficient Primal Simplex Implementation for the Continuous 2-Matching Problem
T H C Smith, T W S Meyer and L Leenen .................................................. 28

Concept Network Framework for a Multi-paradigm Knowledge Base
J Kambanis ........................................... 32

PEST - A Microcomputer Pascal Based Expert System Shell
A G Sartori-Angus and R Neville .......................................................... 39

A Linda Solution to the Evolving Philosophers Problem
S E Hazelhurst ........................................... 44

TECHNICAL NOTE

Knowledge Representation using Formal Grammars
S H von Solms, E M Ehlers and D J Enslin .................................................. 54

COMMUNICATIONS AND REPORTS

Book Review & Books Received ........................................... 57
Cost-effective Visual Simulation Based on Graphics Workstations
A Caduri and D G Kourie .......................................................... 58
A Method of Controlling Quality of Application Software
T D Crossman ........................................... 70
An Update on UNINET-ZA: The Southern African Academic and Research Network
V Shaw ........................................... 75