Malaysia Journal of Computer Science, Vol. 9 No. 1, June 1996, pp. 56-67

AN ATTRIBUTE GRAMMAR APPROACH TO SPECIFYING HALSTEAD'S METRICS

Abdul Azim Ghani
Department of Computer Science
Universiti Pertanian Malaysia
43400 Serdang,
Selangor Darul Ehsan
Malaysia
email: azim@scc.upm.edu.my

Robin Hunter
Department of Computer Science
University of Strathclyde

ABSTRACT

Attribute grammars have been used in defining programming languages and constructing compilers. Since these are concerned with the syntax and static semantics of the source code of the language, attribute grammars can be effectively used to define source code metrics on it. Most of the source code metrics are based on measuring models of the source code. However, there is no formal way of specifying the mapping of the source code onto the models. This paper attempts to provide an approach using an attribute grammar to demonstrate how Halstead's metrics may be specified in an unambiguous manner on the source code itself.

Keywords: attribute grammar, source code models, Halstead's metrics

1.0 INTRODUCTION

Attribute grammars were first proposed by Knuth [1] and have been used as a tool for the formal specification of programming languages. Since then, they have been used in many applications including compiler construction [2], detection of program anomalies [3], as a basis for a language-based editor [4], and as a basis for a software development paradigm [5]. An extended attribute grammar has also been used to define the programming language Pascal [6].

In the field of software metrics, however, most of the source code metrics proposed are normally derived from some sort of models of the source code. For examples, the well-known and widely investigated, McCabe's metric [7] is derived from a flowgraph, a model of control flow structure of the source code, and Halstead's metrics [8] are derived from the counts of tokens found in the source code. In a study by Hunter and Roper [9] in using static analysers to evaluate the metrics, they found that two static analysers produce two different values for the same metrics for a given piece of source code. Although the static analysers are not wrong in the first place, the problem was that there was no precise way by which the source code was mapped onto its respective models, except for a few examples given in the literature.

In this paper we attempt to propose an approach by which Halstead's metrics may be specified in terms of the source code itself, that is by using an attribute grammar formalism. The definition of McCabe's metric using an attribute grammar can be found in Ghani and Hunter [10]. The advantages of using attribute grammars to specify source code metrics are as follows:

- Attribute grammars are as powerful as Chomsky type-0 grammars and are often considered more readable since the context-free grammar on which attribute grammars are based is completely visible.
- Attribute grammars combine the language syntax and static semantics in a modular, declarative fashion, and in a manner from which the static semantic specifications can be readily implemented.
- Attribute grammars have the capability of using techniques for producing efficient parsers directly from them [11].

By exploiting these advantages, we hope we can specify Halstead's metrics in a constructive way from which an implementation can readily be inferred without significant human intervention. In particular we will try to show how the metrics may be specified in an unambiguous manner to aid the metrics collection. Once the metrics are collected, they can be used for examples:

- to predict software product size, cost based on past experiences
- to measure software productivity
- to assess software product quality.

2.0 ATTRIBUTE GRAMMARS

An attribute grammar [1] consists of a context-free grammar (CFG) G = (N,T,P,Z), where N is the set of non-terminal symbols, T is the set of terminal symbols, P is the set of productions, and Z ($Z \in N$) is the start symbol.

With each symbol in the vocabulary V ($V = N \cup T$) of G, there is an associated set of attributes A(X). Each attribute represents a static semantic property of the corresponding symbol. There are two disjoint sets in A(X): I(X) denotes the set of *inherited attributes* of X and S(X) denotes the set of *synthesised attributes* of X.

Inherited attributes are those whose values defined in terms of attributes at the parent and possibly the sibling nodes of node X of the corresponding derivation tree. Synthesised attributes are those whose values are defined in terms of attributes at descendant nodes of node X of the corresponding derivation tree. In our notation a synthesised attribute is preceded by an upward arrow $\uparrow$, and an inherited attribute is preceded by a downward arrow $\downarrow$.

Each of the production $p \in P$ ($p : X_0 \rightarrow X_1X_2...X_n$) of the CFG is augmented with a set of *semantic rules*. Each semantic rule defines how the value of an attribute associated with a terminal or non-terminal in the production is derived by applying a *semantic function* to values associated with other terminals or non-terminals in the production. The semantic rules associated with production p define all the synthesised attributes of the non-terminal symbol X_0 (on the left-hand side of p), as well as all the inherited attributes of symbols $X_1,X_2,...,X_n$ (on the right-hand side of p).

By examining the definition of an attribute grammar and its usage in many applications mentioned before, it is clear that for our purpose the context-free grammar is the context-free means of specifying the syntax of the language and attributes plus semantic functions to specify the evaluation of the source code metrics. In each production of the grammar, the semantic functions specify rules on attribute relationships. These functions are used to determine the values of certain attributes from the values of other attributes which are already known.

3.0 HALSTEAD'S METRICS

Halstead's metrics [8] are among the well-known and widely investigated source code metrics. The basic concern of the metrics is the process of mental manipulation of selecting and arranging program tokens to build a program. These tokens can be either *operators* or *operands*, and a program can be thought of as a sequence of operators and their associated operands. Thus Halstead derived his metrics by first mapping all the program tokens into either operators or operands. Halstead proposed a unified set of metrics that is based on four quantities:

n_1 number of distinct operators in a program or procedure

n_2 number of distinct operands in a program or procedure

N_1 total number of operators in a program or procedure

N_2 total number of operands in a program or procedure

for example the Halstead metrics Vocabulary, Length, and Volume are defined by

$$\text{Vocabulary}(n) = n_1 + n_2$$
$$\text{Length } (N) = N_1 + N_2$$
$$\text{Volume } (V) = N \log_2 n$$

and other Halstead metrics are evaluated similarly. Although these metrics are being discredited according to some publications [12][13], research works related to them are still ongoing [14][15].

One of the problems with these metrics is that although examples are often given in the literature of what constitutes an operator and what constitutes an operand, completely general rules have not been given. In addition it is not always clear what constitute distinct operators and what constitute distinct operands. Since all Halstead's metrics depend on counts of operands and operators, it is important that the counting strategy be clearly defined and consistent [12]. As an example, does the operator symbol '+' occurring twice in a program, once meaning addition of two integers and once meaning addition of two reals, constitute a single operator or two distinct operators? What if one of the plusses had been set addition? The manner in which different counting strategies used have promoted many proposals such as in a commercially available static analysis tool, QUALIGRAPH [16] and in the software analyser distributed by the Purdue University Software Metrics Research Group [17]. Clearly the values of all Halstead's metrics will be affected by the method used to count operators and operands. For this reason the values of the metrics obtained from different tools may not be the same.

The answers to these questions are not really known, and in our definitions of (distinct) operators and operands the assumptions made will be clearly stated, as will our definitions of what constitute operators and operands.

For example, we will assume the following

1) An operand is any symbol which corresponds to a terminal node in the *abstract syntax tree* of a piece of source code. Occurrences of the same identifier to represent different objects (variables, procedure names, types, etc.) will be treated as distinct. Real numbers with the same value e.g., 3.00 and 3.0 will be considered as the same operand, and integers will be treated similarly. The total number of operands in a BLOCK is evaluated by adding the total number of identifiers representing variables and constants in the block to the total number of numbers in a block. The number of distinct operands in a block is evaluated by adding the number of distinct identifiers representing variables and constants in a block to the number of distinct numbers in a block. Note that

many of the terminal symbols appearing in the concrete syntax, e.g., punctuation such as commas, brackets, etc. do not appear, and are not required, in the abstract syntax.

2) An operator is any symbol which corresponds to a non-terminal node of the abstract syntax tree. This includes operator symbols appearing in expressions, as well as 'pseudo' operators such as assignment, while statement, if statement without an else part, if statement with an else part and so on. In particular, a procedure call is regarded as a pseudo operator so that identifiers corresponding to procedure calls will correspond to operators, not operands. Operator symbols which correspond to distinct operators, an operator being identified by the types and number of its operands, will be assumed to represent distinct operators. The number of operators in a BLOCK is given by the number of operator symbols in the block plus the number of pseudo operators in the block. The number of distinct operators in a block is given by the number of operator symbols which correspond to distinct operators, plus the number of distinct pseudo operators in the block.

4.0 SPECIFYING HALSTEAD'S METRICS USING AN ATTRIBUTE GRAMMAR

4.1 Preliminary Requirement

Before considering the steps necessary to specify a method of evaluating Halstead's metrics, we introduce data structures to represent data concerning the occurrences of operands and operators at a particular stage in the analysis of a program in order to evaluate n_1,n_2,N_1,N_2 required to compute Halstead's metrics. There will be three data structures, one concerned with identifiers representing variables and constants (IDS), one concerned with numbers occurring in the program (NUMS), and one concerned with the occurrences of operators and pseudo-operators (OPS). The data structures resemble in some ways a compiler symbol tables used in analysis of source code in order to produce object code.

The data structure IDS is a set of records each of which has the following fields

```
NAME      /* the name of the identifier representing a
             variable or constant*/
TYPE      /* the type of the variable or constant*/
BLOCKLN   /* the block level number of the current
             block*/
BLOCKNO   /* the block number of the current block */
OCCUR     /* the number of applied occurrences of the
             variable or constant*/
OCCBLOCK  /* the number of occurrences in the current
             block*/
```

NAME is the name of an identifier representing a variable or constant declared in the block and TYPE is the variable's type. As in compiler technology BLOCKLN represents the depth of nesting of a block and BLOCKNO its number, the first block entered in analysing the program in a single pass from left to right being numbered one, the second two and so on. OCCUR is the number of applied occurrences (so far) of the variable, and OCCBLOCK the number of occurrences of the variable in the current block.

In order to evaluate the total number of operands and operators, and the number of distinct operands and operators in each block or procedure, as well as for the program as a whole, a number of steps have to be taken.

During analysis of the program a set of records (IDS) will be set up to contain one record (as described above) for each distinct variable or constant in the program. Initially the set will be empty and on the first occurrence of each variable or constant (its declaration or definition) an appropriate record will be set up with the fields as above. At this stage the OCCUR field will be set to zero as will the OCCBLOCK field.

Each applied occurrence of a variable or constant encountered during analysis will cause the corresponding record of IDS to be updated by incrementing the OCCUR and OCCBLOCK fields of the record. A unique record can always be identified for updating from a knowledge of the BLOCKLN and BLOCKNO fields. The analysis will assume the program has no syntax or static semantic errors.

On entering the statement part of each block, the OCCBLOCK field of each record in the set will be set to zero (if it is not already zero)so that at the end of each block the number of times each variable was applied in that block is known. Hence the number of variables in the block and the number of distinct variables in the block are known. Once the analysis is complete IDS will contain a record for each variable and each constant in the program and from these records the total number of distinct variables and constants in the program is simply the number of records in the set, while the total number of variable and constant (occurrences) is obtained by aggregating the values in all the OCCUR fields.
Numbers are represented in the NUMS data structure with elements of the form:

```
NAME      /* the value of the number */
TYPE      /* the type of the number */
OCCUR     /* the number of occurrences of the
             number */
OCCBLOCK  /* the number of occurrences of the
             number in the block */
```

Numbers can be dealt with in a similar way to variables and constants except that they are not declared in any

way, so that the first applied occurrence of a number will cause a record for that number to be set up with the OCCUR and OCCBLOCK fields set to one (rather than zero). Further occurrences of a number for which a record exists will cause the OCCUR and OCCBLOCK fields to be updated. As before, the OCCBLOCK field are set to zero on entering the block. In the case of numbers (unlike identifiers), one record for each number will be sufficient and there is no need for BLOCKLN and BLOCKNO.

At the end of each block the number of numbers and the number of distinct numbers in the block will be available from examination of the OCCBLOCK fields of the records in NUMS, while at the end of the program the corresponding values for the complete program will be known. Assuming that the variables and constants, and the numbers in the program make up all the operands then the number of distinct operands in a block or the complete program is the sum of the number of distinct variables, constants, and numbers in the block or the program. The total number of operands in a block or the complete program is the sum of the total numbers of variables, constants and numbers in the block or program.

Operators are similarly represented by the OPS data structure each element of which is of the form:

NAME	/* the operator symbol or pseudo operator name */
TYPE	/* the types of the operands(where appropriate) */
BLOCKLN	/* the block level number of the current block*/
BLOCKNO	/* the block number of the current block */
OCCUR	/* the number of occurrences of the operator */
OCCBLOCK	/* the number of occurrences of the operator in the block*/

Each conventional (as opposed to pseudo) operator has a type associated with it (the types of its operands) and two operators are distinct if the types (or number) of their operands are distinct. In addition, procedure names associated with procedure pseudo operators will have a BLOCKLN and BLOCKNO associated with them reflecting the scope rules of the language. Subject to this, operators can be treated in the same way as numbers and the numbers of distinct operators in a block or the complete program can be obtained at the end of the block or the end of the program from the elements in the OPS. The total number of operators in a block or program may be obtained similarly.

The steps outlined above are the principal actions used in the attribute grammar to follow, in order to DEFINE the metrics. They are not parts of rules for EVALUATING the metrics and are too informal to be used directly in their definition. However they are intended for use along with the attribute grammar which we will describe to help understand the rather formal type of definition discussed next. If the above rules, as described, appear in any way to conflict with the attribute grammar definition, then the latter should be preferred in all cases.

4.2 Definition of Halstead's Metrics

The metrics will be defined in terms of the subset of the Pascal language. We will use Backus Naur Form (BNF) for productions of the language's context-free grammar in the attribute grammar with appropriate attributes related to the metrics evaluation are augmented. Non-terminals in the grammar are enclosed in angle brackets, "< >". The symbol "::=" is used to separate the left and right part of a production. Terminals are represented either by lower case Roman characters or in the case of special characters, the characters are enclosed in a single quotation marks. Attributes variables are represented by upper case Roman characters, a synthesised attribute is preceded by an upward arrow, "↑", and an inherited attribute is preceded by a downward arrow, "↓". The arrow indicates the direction of travel of attribute value. The name of attribute variable chosen as far as possible, will reflect the meaning of the attribute. The inclusion of a digit or the alphabet U or Z at the end of each attribute variable, or the alphabet S in front of each attribute variable of the same name distinguishes the attribute variables of the same set. Where an attribute variable appears two or more times in a production, each occurrence will have the same value. Rules to specify the relationships between attributes are enclosed in square bracket, "[ ]", and are embedded in the part of a production to which they apply.

What follows is not a complete definition but it is sufficient to specify how a complete definition could be obtained. A detail explanation about functions used in the definition can be found in Appendix I. The work of Watt [6] in using an extended attribute grammar to define Pascal is acknowledged.

i. Program and block

<program> ↑IDS ↑OPS ↑NUMS
::= <progheading> ↑BLOCKTAB ↑BLOCKLN ↑BLOCKNO
';'
<block> ↓{} ↓{} ↓{} ↓BLOCKTAB ↓BLOCKLN ↓BLOCKNO ↑IDS ↑OPS ↑NUMS
'.'

[comments: ↑IDS, ↑OPS and ↑NUMS are the three sets containing, on completion of the analysis, the full data on all the variables, operators and numbers in the program. They are synthesised in this rule. The three sets may be used to obtain all the Values of

n_1,n_2,N_1 and N_2 both for the complete program and for the main block. As far as this rule is concerned empty sets of identifiers, operators and numbers are inherited].

The corresponding production and rules for <progheading> are fairly simple:

```
<progheading> ↑BLOCKTAB ↑BLOCKLN ↑BLOCKNO
        ::= program identifier ↑BLOCKTAB
                               ↑BLOCKLN ↑BLOCKNO
        [rules:  ↑BLOCKLN = 0
                 ↑BLOCKNO = 0
                 ↑BLOCKTAB = setup_block_table ]
```

showing where the block level number, the block number and the block table are initialised.

The production and rules for block are however complex mainly due to the fact that identifiers can be declared/defined in a number of places.

```
<block> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB ↓BLOCKLN
        ↓BLOCKNO ↑SIDS ↑SOPS ↑SNUMS ↑CBNO
        ::=  <constdec> ↓IDSZ ↓BLOCKLN
                        ↓BLOCKNO ↑IDS1
             <vardec> ↓IDS1 ↓BLOCKLN ↓BLOCKNO
                      ↑IDS2
             <prodecs> ↓IDS2  ↓OPSZ ↓NUMSZ
                       ↓BLOCKTAB ↓BLOCKLN
                       ↓BLOCKNO ↓BLOCKNO ↑IDS3
                       ↑OPS3 ↑NUMS3 ↑CBNO
             <stmpart> ↓IDS3 ↓OPS3 ↓NUMS3
                       ↓BLOCKTAB ↓BLOCKLN
                       ↓BLOCKNO ↑SIDS ↑SOPS
                       ↑SNUMS
         [rules: ↓IDSZ ↓OPSZ and ↓NUMSZ are ↓IDS
         ↓OPS and ↓NUMS respectively with the
         OCCBLOCK fields of all their elements set to zero ]
```

A block has five inherited attributes

- set of declarations of variables and constants for complete program(so far) ↓IDS
- set of operators for the complete program(so far) ↓OPS
- set of numbers for the complete program (so far) ↓NUMS

along with ↓BLOCKLN and ↓BLOCKNO.

There are also three synthesised attributes ↑SIDS ↑SOPS ↑SNUMS and ↑CBNO from the first three of which the values for n_1,n_2,N_1 and N_2 for the block may be computed.

The various parts of a block inherit appropriate attributes and synthesis others. In this way a block within the block will itself be able to inherit appropriate attributes in order that it may synthesise the values necessary to evaluate Halstead's metrics for the block itself and the complete program.

ii. Constant declarations

The productions specify the inclusion of any new constant identifiers declared in a particular block (program or procedure) to the set of declaration of variables and constants ↑IDS1. The *addel* function does this inclusion.

```
<constdec> ↓IDSZ ↓BLOCKLN ↓BLOCKNO ↑IDS1
        ::= <empty> ↓IDSZ ↑IDS1
        [rule: ↑IDS1 = ↓IDSZ ]
        | 'const' <constseqs> ↓IDSZ ↓BLOCKLN
                              ↓BLOCKNO ↑IDS1

<constseqs> ↓IDSZ ↓BLOCKLN ↓BLOCKNO ↑IDS1
        ::= <constseq> ↓IDSZ ↓BLOCKLN ↓BLOCKNO
                       ↑IDS1 ';'
        | <constseqs> ↓IDSZ ↓BLOCKLN ↓BLOCKNO
                      ↑IDSS
          <constseq> ↓IDSS ↓BLOCKLN ↓BLOCKNO
                     ↑IDS1 ';'

<constseq> ↓IDSZ ↓BLOCKLN ↓BLOCKNO  ↑IDSU
        ::= identifier  ↓IDSZ ↓BLOCKLN ↓BLOCKNO
                        ↑IDSU ↑NAME
            '=' <constant> ↑TYPE
        [rule:  ↑IDSU = addel ( ↓IDSZ, ↑NAME, ↑TYPE,
                                ↓BLOCKLN,  ↓BLOCKNO) ]
```

iii. Variable declarations

Any new variable declared in a block is added to set of declarations of variables and constants. The productions below specify this inclusion. Again *addel* function does the inclusion.

```
<vardec>  ↓IDS1 ↓BLOCKLN ↓BLOCKNO ↑IDS2
        ::= <empty>  ↓IDS1 ↑IDS2
        [rule: ↑IDS2 =  ↓IDS1 ]
        | var <vardefns>  ↓IDS1 ↓BLOCKLN
                          ↓BLOCKNO ↑IDS2

<vardefns>  ↓IDS1 ↓BLOCKLN ↓BLOCKNO ↑IDS2
        ::= <vardefn> ↓IDS1 ↓BLOCKLN ↓BLOCKNO
                      ↑IDS2 ';'
        | <vardefns>  ↓IDS1 ↓BLOCKLN ↓BLOCKNO
                      ↑IDSS
          <vardefn>  ↓IDSS ↓BLOCKLN ↓BLOCKNO
                     ↑IDS2 ';'

<vardefn> ↓IDS1 ↓BLOCKLN ↓BLOCKNO ↑IDS2
        ::= <varlist> ↓IDS1 ↓BLOCKLN ↓BLOCKNO
                      ↓TYPE ↑IDS2
          ':' <typedenoter> ↑TYPE

<varlist> ↓IDS1 ↓BLOCKLN ↓BLOCKNO ↓TYPE ↑IDS2
        ::= identifier ↑NAME
```

```
        [rule: ↑IDS2 = addel(↓IDS1, ↑NAME, ↓TYPE,
                    ↓BLOCKLN, ↓BLOCKNO) ]
        | <varlist> ↓IDS1 ↓BLOCKLN ↓BLOCKNO
                    ↓TYPE ↑IDSD
          ',' identifier ↓IDSD ↑NAME ↑IDS2
          [rule: ↑IDS2 = addel(↓IDSD, ↑NAME, ↓TYPE,
                              ↓BLOCKLN, ↓BLOCKNO) ]
```

iv. Procedure Declarations

In order to show how the final values for the metrics in a block are obtained we should consider the productions and rules for <prodecs>. There are two productions to be considered.

```
<prodecs> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
        ↓BLOCKLN ↓BLOCKNO ↓OBN ↑SIDS ↑SOPS
        ↑SNUMS ↑CBNO
        ::= <empty> ↓IDS ↓OPS ↓NUMS ↓BLOCKLN
                    ↓BLOCKNO ↑SIDS ↑SOPS
                    ↑SNUMS ↑CBNO
          [rules:  ↑SIDS = ↓IDS
                   ↑SOPS = ↓OPS
                   ↑SNUMS = ↓NUMS
                   ↑CBNO = ↓BLOCKNO ]
         | <proc> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                   ↓BLOCKLN ↓BLOCKNO ↓OBNO
                   ↑IDS1 ↑OPS1 ↑NUMS1 ↑CBNO1 ';'
           <prodecs> ↓IDS1 ↓OPS1 ↓NUMS1
                   ↓BLOCKTAB ↓BLOCKLN
                   ↓CBNO1 ↓OBNO ↑SIDS ↑SOPS
                   ↑SNUMS ↑CBNO
```

Note that the current block number ↑CBNO1 is synthesised from <proc> and inherited by <prodecs>. In a similar way ↓BLOCKNOS is inherited from <procheading> in the following rule. The block level number is reset 'automatically' on returning to a level.

```
<proc> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB ↓BLOCKLN
       ↓BLOCKNO ↓OBNO ↑BIDS ↑BOPS ↑BNUMS
       ↑CBNO
         ::= <procheading> ↓OPS ↓BLOCKLNS
                           ↓BLOCKNOS ↓BLOCKLN
                           ↓OBNO ↑OPS1 ↑FORM
           <block> ↓FIDS ↓OPS1 ↓NUMS
                   ↓BLOCKTABU ↓BLOCKLNS
                   ↓BLOCKNOS ↑BIDS ↑BOPS ↑BNUMS
                   ↑CBNO
         [rules:   ↓FIDS = union_of (↓IDS, ↑FORM)
                   ↓BLOCKLNS = ↓BLOCKLN + 1
                   ↓BLOCKNOS = ↓BLOCKNO + 1
                   ↓BLOCKTABU =
                   update_block_table(↓BLOCKLNS,
                   ↓BLOCKNOS, ↓BLOCKTAB,↓OBNO) ]
```

showing (amongst other things) where the block level number and the block number are updated. ↑FORM is the set of formal parameters for the procedure.

Some more productions and rules:

```
<procheading> ↓OPS ↓BLOCKLNS ↓BLOCKNOS
                ↓BLOCKLN ↓OBNO ↑OPS1 ↑FORM
        ::=  'proc' procedure_identifier ↑NAME
           <formalpar> ↓BLOCKLNS ↓BLOCKNOS
                       ↑FORM
        [rule: ↑OPS1 = addel(↓OPS, ↑NAME,
                           ↓BLOCKLN, ↓OBNO) ]
```

addel is a function to update the set operators. In this case a new pseudo operator is added to the operator set.

The productions specified next show the formation of the set of formal parameter ↑FORM in a procedure which is synthesised to <procheading>.

```
<formalpar> ↓BLOCKLNS ↓BLOCKNOS ↑FORM
        ::= <empty>
          [rule: ↑FORM = {} ]
        | '(' <formalparlist> ↓BLOCKLNS ↓BLOCKNOS
                           ↑FORM ')'

<formalparlist> ↓BLOCKLNS ↓BLOCKNOS ↑FORM
        ::= <formalparsec> ↓BLOCKLNS ↓BLOCKNOS
                           ↑FORM
        | <formalparlist> ↓BLOCKLNS ↓BLOCKNOS
                          ↑FORM1 ';'
          <formalparsec> ↓BLOCKLNS ↓BLOCKNOS
                          ↑FORM2
          [rules ↑FORM = ↑FORM1 + ↑FORM2 (where
                '+' means set addition) ]

<formalparsec> ↓BLOCKLNS ↓BLOCKNOS ↑FORM
        ::= <valuepart> ↓BLOCKLNS ↓BLOCKNOS
                        ↑FORM
        | <varpart> ↓BLOCKLNS ↓BLOCKNOS ↑FORM

<valuepart> ↓BLOCKLNS ↓BLOCKNOS ↑FORM
        ::= <vardefn> ↓{} ↓BLOCKLNS ↓BLOCKNOS
                      ↑FORM

<varpart> ↓BLOCKLNS ↓BLOCKNOS ↑FORM
        ::= 'var' <vardefn> ↓{} ↓BLOCKLNS
                            ↓BLOCKNOS ↑FORM
```

v. Statements

Productions for the various statements and their associated rules are merely involved in passing attributes from left to right (down the syntax tree) or from right to left (up the syntax tree).

```
<stmpart> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
          ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
          ↑SNUMS
          ::= <compoundstat> ↓IDS ↓OPS ↓NUMS
                             ↓BLOCKTAB ↓BLOCKLN
                             ↓BLOCKNO ↑SIDS ↑SOPS
                             ↑SNUMS
```

```
<compoundstat> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                    ↓BLOCKLN ↓BLOCKNO ↑SIDS
                    ↑SOPS ↑SNUMS
          ::= 'begin'
              <stmtseq> ↓IDS ↓OPS1 ↓NUMS
                          ↓BLOCKTAB ↓BLOCKLN
                          ↓BLOCKNO ↑SIDS ↑SOPS
                          ↑SNUMS
             'end'
          [rule:    ↓OPS1 = updateoperator(↓OPS, csop) ]
```

whereas in some cases have to be passed up and down adjacent branches of the syntax tree:

```
<stmtseq> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
            ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
            ↑SNUMS
          ::= <statement> ↓IDS ↓OPS ↓NUMS
                          ↓BLOCKTAB ↓BLOCKLN
                          ↓BLOCKNO ↑SIDS ↑SOPS
                          ↑SNUMS
          | <statement> ↓IDS ↓OPS ↓NUMS
                          ↓BLOCKTAB ↓BLOCKLN
                          ↓BLOCKNO ↑IDSS ↑OPSS
                        ↑NUMSS
          ';'
          <stmtseq> ↓IDSS ↓OPSS ↓NUMSS
                      ↓BLOCKTAB ↓BLOCKLN
                      ↓BLOCKNO ↑SIDS ↑SOPS ↑SNUMS
```

There are a number of different types of statement:

```
<statement> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
              ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
              ↑SNUMS
          ::=  <empty> ↓IDS ↓OPS ↓NUMS ↓BLOCKLN
                       ↓BLOCKNO ↑SIDS ↑SOPS
                       ↑SNUMS
          [rules:    ↑SIDS = ↓IDS
                     ↑SOPS = ↓OPS
                     ↑SNUMS = ↓NUMS ]
          | <compoundstat> ↓IDS ↓OPS ↓NUMS
                           ↓BLOCKTAB ↓BLOCKLN
                           ↓BLOCKNO ↑SIDS ↑SOPS
                           ↑SNUMS
          | <assignstat> ↓IDS ↓OPS ↓NUMS
                         ↓BLOCKTAB ↓BLOCKLN
                         ↓BLOCKNO ↑SIDS ↑SOPS
                         ↑SNUMS
          | <structstat> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                         ↓BLOCKLN ↓BLOCKNO ↑SIDS
                         ↑SOPS ↑SNUMS
          | <procstat> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                       ↓BLOCKLN ↓BLOCKNO ↑SIDS
                       ↑SOPS ↑SNUMS
```

In an <assignstat> the ':=' operator is recognised and dealt with

```
<assignstat> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
               ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
               ↑SNUMS
          ::= <varaccess> ↓IDS ↓BLOCKTAB
                          ↓BLOCKLN ↓BLOCKNO
                          ↑IDS1 ↑TYPE
             ':='
             <expression> ↓IDS1 ↓OPS1 ↓NUMS
                          ↓BLOCKTAB ↓BLOCKLN
                          ↓BLOCKNO ↑SIDS ↑SOPS
                          ↑SNUMS ↑TYPE
           [rule: ↓OPS1 = updateoperator(↓OPS,
                                         assignop) ]
```

where *updateoperator* performs the following:

if a record for the assignment operator is not already in OPS, a suitable record is created and added to the set with the OCCUR field set to one and the OCCBLOCK field set to one. Otherwise (a record for the assignment operator is already in OPS) the OCCUR and OCCBLOCK fields of the record are incremented as appropriate.

<structstat> that contains <condstat> in which it specifies the if statement, and <whilestat> statements are dealt with similarly

```
<structstat> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
               ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
               ↑SNUMS
          ::= <condstat> ↓IDS ↓OPS ↓NUMS
                         ↓BLOCKTAB ↓BLOCKLN
                         ↓BLOCKNO ↑SIDS ↑SOPS
                         ↑SNUMS
          | <whilestat> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                        ↓BLOCKLN ↓BLOCKNO ↑SIDS
                        ↑SOPS ↑SNUMS
```

and the while statement will serve as an example

```
<whilestat> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
              ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
              ↑SNUMS
          ::=  'while'
             <condition> ↓IDS ↓WWOPS ↓NUMS
                         ↓BLOCKTAB ↓BLOCKLN
                         ↓BLOCKNO ↑WIDS ↑WOPS
                         ↑WNUMS
             'do'
             <statement> ↓WIDS ↓WOPS ↓WNUMS
                         ↓BLOCKTAB ↓BLOCKLN
                         ↓BLOCKNO ↑SIDS ↑SOPS
                         ↑SNUMS
          [rule: ↓WWOPS = updateoperator (↓OPS,
                                          whiledoop) ]
```

where

```
<condition> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
              ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
              ↑SNUMS
          ::= <expression> ↓IDS ↓OPS ↓NUMS
                           ↓BLOCKTAB ↓BLOCKLN
```

```
                        ↓BLOCKNO ↑SIDS ↑SOPS
                        ↑SNUMS ↑TYPE
```

The <procstat> which is the procedure call statement can be dealt with as follows:

```
<procstat> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
           ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
           ↑SNUMS
         ::= identifier ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                        ↓BLOCKLN ↓BLOCKNO ↑SIDS
                        ↑SOPS ↑SNUMS ↑NAME
          [rules: ↑SOPS = updateoperator(↓OPS,
                          ↑NAME, ↓BLOCKLN,
                          ↓BLOCKNO,↓BLOCKTAB)
                  ↑SIDS = ↓IDS
                  ↑SNUMS = ↓NUMS ]
         | identifier ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                      ↓BLOCKLN ↓BLOCKNO ↑SIDS
                      ↑SOPS ↑SNUMS ↑NAME
          '(' <exprlist> ↓IDS ↓POPS ↓NUMS
                         ↓BLOCKTAB ↓BLOCKLN
                         ↓BLOCKNO ↑SIDS ↑SOPS
                         ↑SNUMS ')'
          [rule: ↓POPS = updateoperator(↓OPS, ↑NAME,
                                 ↓BLOCKLN, ↓BLOCKNO,
                                 ↓BLOCKTAB) ]
```

where *updateoperator* in this case adds to its parameter list the procedure name and scope information ↓BLOCKLN and ↓BLOCKNO as well as ↓BLOCKTAB in order to differentiate different procedure calls as distinct operators.

vi. Expressions

The productions and rules for expressions seem quite complex but, apart from updating the number of occurrences of the operators involved AND passing attributes down and up the syntax tree, are fairly straightforward. The following productions specify some of them.

```
<expression> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
             ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
             ↑SNUMS ↑TYPE
           ::= <simpleexpr> ↓IDS ↓OPS ↓NUMS
                            ↓BLOCKTAB ↓BLOCKLN
                            ↓BLOCKNO ↑SIDS ↑SOPS
                            ↑SNUMS ↑TYPE
           | <simpleexpr> ↓IDS ↓OPS ↓NUMS
                          ↓BLOCKTAB ↓BLOCKLN
                          ↓BLOCKNO ↑SDS ↑SPS
                          ↑SUMS ↑TYPE1
            '=' <simpleexpr> ↓SDS ↓ESPS ↓SUMS
                             ↓BLOCKTAB ↓BLOCKLN
                             ↓BLOCKNO ↑SIDS ↑SOPS
                             ↑SNUMS ↑TYPE2
             [rules:  ↓ESPS = updateoperator(↑SPS, 'EQ',
                                  ↑TYPE1, ↑TYPE2)
                      ↑TYPE = result_of('EQ', ↑TYPE1,
                                  ↑TYPE2) ]

<simpleexpr> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
             ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
             ↑SNUMS ↑TYPE
           ::= <term> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                      ↓BLOCKLN ↓BLOCKNO ↑SIDS
                      ↑SOPS ↑SNUMS ↑TYPE
           | <sign> ↑NAME
             <term> ↓IDS ↓NOPS ↓NUMS ↓BLOCKTAB
                    ↓BLOCKLN ↓BLOCKNO ↑SIDS
                    ↑SOPS ↑SNUMS ↑TYPE
            [rule: ↓NOPS = updateoperator(↓OPS, ↑NAME,
                                  ↑TYPE) ]
           | <simpleexpr> ↓IDS ↓OPS ↓NUMS
                          ↓BLOCKTAB ↓BLOCKLN
                          ↓BLOCKNO ↑SDS ↑SPS
                          ↑SUMS ↑TYPE1
            '+' <term> ↓SDS ↓ESPS ↓SUMS ↓BLOCKTAB
                       ↓BLOCKLN ↓BLOCKNO ↑SIDS
                       ↑SOPS ↑SNUMS ↑TYPE2
             [rules:   ↓ESPS = updateoperator(↑SPS, 'PLUS',
                                  ↑TYPE1, ↑TYPE2)
                       ↑TYPE = result_of('PLUS', ↑TYPE1,
                                  ↑TYPE2) ]

<term> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB ↓BLOCKLN
       ↓BLOCKNO ↑SIDS ↑SOPS ↑SNUMS ↑TYPE
         ::= <factor> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                      ↓BLOCKLN ↓BLOCKNO ↑SIDS
                      ↑SOPS ↑SNUMS ↑TYPE
         | <term> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
                  ↓BLOCKLN ↓BLOCKNO ↑SDS ↑SPS
                  ↑SUMS ↑TYPE1
          '*' <factor> ↓SDS ↓ESPS ↓SUMS
                       ↓BLOCKTAB ↓BLOCKLN
                       ↓BLOCKNO ↑SIDS ↑SOPS
                       ↑SNUMS ↑TYPE2
           [rules: ↓ESPS = updateoperator(↑SPS, 'TIMES',
                                  ↑TYPE1, ↑TYPE2)
                   ↑TYPE = result_of ('TIMES', ↑TYPE1,
                                  ↑TYPE2) ]
```

in which the relational operator '=', the arithmetic operators '+' and '*' are updated in the set of operators using the *updateoperator* function. In these cases the types of the simple expressions are included as parameters to the function in order to fully identify the operator concerned. The other relational and arithmetic operators can be dealt with similarly. However in the case of monadic operator, ↑NAME which has a value either PLUS or MINUS and the type of its term are used as the different parameters to the function *updateoperator*.

We now have a look at the mechanisms for updating occurrences of operands which occur around the terminal nodes of the syntax tree. We look at how occurrences of numbers and variables are updated.

A factor may (amongst other things) be a constant or a variable access, the rule being

```
<factor> ↓IDS ↓OPS ↓NUMS ↓BLOCKTAB
         ↓BLOCKLN ↓BLOCKNO ↑SIDS ↑SOPS
         ↑SNUMS ↑TYPE
         ::= <varaccess> ↓IDS ↓BLOCKTAB
                         ↓BLOCKLN ↓BLOCKNO
                         ↑SIDS ↑TYPE
           [rules:  ↑SOPS = ↓OPS
                    ↑SNUMS = ↓NUMS ]
         | <constant> ↓NUMS ↓BLOCKLN ↓BLOCKNO
                      ↑SNUMS ↑TYPE
         [rule:     ↑SIDS = ↓IDS
                    ↑SOPS = ↓OPS ]
```

A variable access <varaccess> in turn is an identifier

```
<varaccess> ↓IDS ↓BLOCKTAB ↓BLOCKLN
            ↓BLOCKNO ↑SIDS ↑TYPE
         ::= identifier ↑NAME ↑TYPE
          [rule:    ↑TYPE = findtype(↑NAME, ↓IDS,
                              ↓BLOCKLN, ↓BLOCKNO,
                              ↓BLOCKTAB)
                    ↑SIDS = updateids(↓IDS, ↑NAME,
                              ↓BLOCKLN, ↓BLOCKNO,
                              ↓BLOCKTAB)  ]
```

where *updateids* will increment the OCCUR and OCCBLOCK fields of the element of ↓IDS identified by ↓NAME, ↓BLOCKLN, ↓BLOCKNO by one.

A constant in turn, could be a positive integer or a real number

```
<constant> ↓NUMS ↓BLOCKLN ↓BLOCKNO ↑SNUMS
           ↑TYPE
         ::= <intreal> ↓NUMS ↓BLOCKLN ↓BLOCKNO
                       ↑SNUMS ↑TYPE
```

and an intreal may be an integer

```
<intreal> ↓NUMS ↓BLOCKLN ↓BLOCKNO ↑SNUMS
          ↑TYPE
         ::= number ↓NUMS ↓BLOCKLN ↓BLOCKNO
                    ↑SNUMS ↑VALUE ↑TYPE
         [rules:  ↑TYPE = 'INTEGER'
                  ↑SNUMS = updatenums (↓NUMS,
                                ↑VALUE, ↑TYPE) ]
```

where *updatenums* will create a new element in the set NUMS (if none already exists) with fields OCCUR and OCCBLOCK set to one and the other fields set appropriately or, if a record for the constant is already in the set, then the OCCUR and OCCBLOCK fields are each incremented by one.

5.0 EVALUATION OF THE DEFINITION

From the above definition, an evaluation tool for Halstead's metrics can be produced using the Unix compiler building tools Lex and YACC [18]. Lex is a lexical analyser generator and YACC is a bottom-up SLR(1) parser-generator. Lex is used to write a lexical analyser for the definition that recognizes the regular expressions which match the specific tokens in the grammar. YACC is used to construct the parser that calls the lexical analyser to produce the next token. The construction is based on the context-free grammar provided by the definition. The parser recognizes a sequence of matching tokens described by the grammar.

YACC also allows rules to evaluate the metrics in the grammar to be transformed to actions written in the language C. The actions are inserted at appropriate productions in the parser. The actions may make use of variables of the form $n (where n is an integer) and $$ which can be used to represent attributes of symbols in a production. The convention being that the variable $$ is associated with the symbol on the left hand side of the production and $n is associated with the n'th symbol on the right hand side of the production. However YACC only allows one synthesised attribute in the production to be represented by the variable $$. Thus a production with more than one attributes cannot be represented directly in YACC. In order to overcome this characteristic, global variables are used to represent the attributes. Likewise when more than one attribute names are used to represent attributes of the same type, a global variable is used to represent the attributes.

If the evaluation of the four parameters on which the Halstead's metrics are based was to be performed by a literal implementation of the definition, a major activity would involve passing sets of declarations, numbers and operators around various parts of the underlying context-free grammar. The cost of copying the sets are prohibitive and to avoid it, global symbol tables representing the sets are used. The symbol tables are not copied as the parser moves through the grammar, but the global versions of them are kept all the time in order that the values of n_1,n_2,N_1 and N_2 for the complete program may be computed.

As an example the following part of the definition to specify the inclusion of any new constant identifier declared in a particular block to the set of declaration of variables and constants

```
<constseq> ↓IDSZ ↓BLOCKLN ↓BLOCKNO  ↑IDSU
         ::= identifier  ↓IDSZ ↓BLOCKLN ↓BLOCKNO
                         ↑IDSU ↑NAME
            '=' <constant> ↑TYPE
           [rule:  ↑IDSU = addel ( ↓IDSZ, ↑NAME, ↑TYPE,
                                   ↓BLOCKLN,  ↓BLOCKNO) ]
```

in YACC becomes

```
constseq    : IDENTIFIER '=' constant
            {
              idset = addel(idset,$1,$3,blockln,blockno)
            }
```

idset is the global symbol table for the set of declaration of variables and constant, *blockln* is the global variable for the block level number and *blockno* is the global variable for the block number. $1 holds the constant identifier name and $3 holds the type of the constant. The transformation of other parts of the definition to YACC can be dealt with similarly.

6.0 CONCLUSIONS

An attribute grammar approach to specifying source code metrics has been presented in this paper by defining, well-known source code metrics, Halstead's metrics. This approach is an attempt to use the relatively mature theory of programming languages to define source code metrics. Halstead's metrics have been used to illustrate the approach since they are typical and the more complex source code metrics currently in use and have long suffered from the lack of a precise definition. An important advantage of the approach is the fact that it could lead to well-defined source code metrics, in contrast to model-based approach which can suffer from lack of a precise definition of how the source code maps on to the model.

The discussion presented in this paper shows that this approach has a role in providing more rigorous definitions of source code metrics which can aid the evaluation of the source code metrics. Attribute grammars have been used to define programming languages rigorously and used to construct compilers. The rigour and ease of implementation of attribute grammars strongly suggest their use to define source code metrics, and as a basis for the construction of compiler like tools (or even better compiler extension) to evaluate them.

Specific to this paper is the definition of Halstead's metrics for a subset of Pascal language which only uses integer and real data types. The definition could be extended to include other data type but with some modification to the attribute grammar to represent the data type. The idea of using attribute grammars to define source code metrics could also be extended to other programming language paradigms, such as logical programming languages, functional programming languages, or object oriented programming languages, if their grammars are defined in terms of attribute grammars. Some of the attributes involved in the grammars could be associated with metrics evaluation. However there should be some changes to the way we used in this paper since there are some characteristic of the languages that are unique to the type of the languages.

REFERENCES

[1] D. E. Knuth, "Semantics of Context-free languages". *Mathematical System Theory*, Vol. 2, No. 2, 1968, pp. 127-145.

[2] P. M. Lewis, D. J. Rosenkrantz and R. E. Stearns, "Attribute Translations". *Journal of Computing Systems and Science*, Vol. 9, No. 3, 1974, pp. 279-307.

[3] L. J. Arthur and J. Ramanathan, "Design of Analyzers for Selective Program Analysis". *IEEE Transactions on Software Engineering*, Vol. 7, No. 1, 1981, pp. 39-51.

[4] T. Reps, *Generating Language-based Environments*. Cambridge, Massachusetts, MIT Press, 1984.

[5] R. A. Frost, "Constructing Programs as Executable Attribute Grammars". *The Computer Journal*, Vol. 35, No. 4, 1992, pp. 376-389.

[6] D. A. Watt, "An Extended Attribute Grammar for Pascal". *ACM Sigplan Notices*, Vol. 14, No. 2, 1979, pp. 60 -74.

[7] T. McCabe, "A complexity Measure". *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, 1976, pp. 308-320.

[8] M. Halstead, *Elements of Software Science*. Amsterdam: Elsevier North-Holland, 1977.

[9] R. B. Hunter and R. M. F. Roper, "Standardisation of Source Code Metrics", in *EUROMETRICS 91 Conference, Paris, 1991.*

[10] A. A. Ghani and R. B. Hunter, "Standards for Source Code Metrics", in *Proceedings of the Eleventh Conference of the South East Asia Regional Computer Confederation, Kuala Lumpur, August 1992*, pp. 40.01-40.15.

[11] F. L. DeRemer, "Simple LR(k) Grammars". *Communications of the ACM*, Vol. 14, No. 7, 1971, pp. 453-460.

[12] M. A. Lister, "Software Science - The Emperor's New Clothes?". *Australian Computer Journal*, Vol. 14, No. 2, 1982, pp. 66-71.

[13] P. G. Hamer and G. D. Frewin, "M. H. Halstead's Software Science - A Critical Examination", in *Proceedings of 6th International Conference of Software Engineering, IEEE, 1982*, pp. 197-206.

[14] Y. S. Chen, "Zipf-Halstead Theory of Software Metrication". *International Journal of Computer Mathematics*, Vol. 14, No. 3/4, 1992, pp. 125-138.

[15] S. Keller-McNulty, M. S. McNulty and D. A. Gustafson, "Stochastic-Models for Software Science". *Journal of System and Software*, Vol. 16, No. 1, 1991, pp. 59-68.

[16] S. Szentes, *QUALIGRAPH User Guide*. Budapest, Computer Research and Innovation Centre, 1986.

[17] S. D. Conte, H. E. Dunsmore and V. Y. Shen, Software Engineering Metrics and Models. Menlo Park, California: Benjamin Cummings, 1986.

[18] T. Mason and D. Brown, *Lex and Yacc*. Sebastopol, California, O'Reilly and Associate Inc., 1990.

APPENDIX I

Some of the functions which appear require a brief explanation

addel(↓IDS, ↑NAME, ↓TYPE, ↓BLOCKLN, ↓BLOCKNO)
addel(↓OPS, ↑NAME, ↓BLOCKLN, ↓BLOCKNO)

causes a new element of an identifier set IDS or an operator set OPS to be formed. In the first case the new element of IDS is required for a constant or variable declaration and the fields of the new element are assigned as follows:

↑NAME - the name of the identifier representing the constant or variable
↓TYPE - the type of the constant or variable
↓BLOCKLN - the block level where it is declared
↓BLOCKNO - the block number in which it is declared

the OCCUR and OCCBLOCK fields of the element being initialised to zero.

The second case corresponds to a new element of ↓OPS being required for each procedure declaration. In this case there is no ↓TYPE field otherwise the fields of the set element are initialised in the same way.

updateoperator(↓OPS, pseudo-operator)
updateoperator(↓OPS, operator, ↑TYPE) for monadic operator
updateoperator(↓OPS, operator, ↑TYPE1, ↑TYPE2) for dyadic operator
updateoperator(↓OPS, ↑NAME, ↓BLOCKLN, ↓BLOCKNO, ↓BLOCKTAB)

updateoperator takes several forms, one for pseudo-operators, such as statements , with two parameters; one for conventional operators such as arithmetic and set operators with three or four parameters; and one for procedure calls also with four parameters. In each case the first parameter is the operator set which is to be enhanced. The occurrence of a pseudo operator requires one more parameter, the name of the pseudo operator. The occurrence of an arithmetic, relational or set operator requires a further one (for a monadic operator) or two (for a dyadic operator) parameters being the parameter type(s) of the operands required to identify fully the operator concerned. In order that calls of different procedures should be recognised as distinct operators, the procedure name and scope information (↓BLOCKLN, ↓BLOCKNO, ↓BLOCKTAB) are required to be parameters of the call. The first three forms may also be used to create a new element of ↓OPS where none exists.

updateids(↓IDS, ↑NAME, ↓BLOCKLN, ↓BLOCKNO, ↓BLOCKTAB)

will increment the OCCUR and OCCBLOCK fields of the element of ↓IDS identified by the fields ↑NAME, ↓BLOCKLN, ↓BLOCKNO.

updatenums (↓NUMS, ↑VALUE, ↑TYPE)

is similar to *updateoperator* though it only takes one form with a single type parameter. The values of ↑VALUE and ↑TYPE identify the appropriate element of ↓NUMS, if one exists, to have its OCCUR and OCCBLOCK fields incremented by one. If no element of ↓NUMS exists with the given ↑VALUE and ↑TYPE fields, a new element is created with its OCCUR and OCCBLOCK fields set to one.

findtype(↑NAME, ↓IDS, ↓BLOCKLN, ↓BLOCKNO, ↓BLOCKTAB)

finds the type associated with an identifier corresponding to its current scope from ↓IDS with the aid of the *block table*.

setup_block_table

delivers the initialised *block table*.

update_block_table(↓BLOCKLNS, ↓BLOCKNOS, ↓BLOCKTAB,↓OBNO)

delivers the updated *block table*.

form_number(↑VAL1, ↑VAL2)

forms a real number where ↑VAL1 represents the digits before the point and ↑VAL2 represents the digits after the point.

result_of(OP, ↑TYPE1, ↑TYPE2)

whose value is the type of the result of applying the dyadic operator OP

union_of

simply forms the union of the two sets given as parameters.
List of pseudo operators:

assignop
csop
dotdotop
ifthenop
ifthenelseop
whiledoop

BIOGRAPHY

Abdul Azim Ghani is the Deputy Director of Computer Centre cum a lecturer at the Department of Computer Science, Universiti Pertanian Malaysia. He obtained his Ph.D in Computer Science from Strathclyde University in 1993. His research interests include software metrics, software engineering, software product and software process assessment and improvement.

Robin Hunter is a Senior Lecturer at Strathclyde University. His research interests include software product and software process assessment and improvement.