SETL Newsletter # 141

J.T. Schwartz
January 16, 1975

Reflections on some very-high level Dictions having an English / "Automatic Programming" Flavor.

1. Conventions which make 'implicit references' available.

Nouns appearing in English-language discourse are typically very highly 'typed', i.e., are known to refer to objects of known 'types' and thus to appear only in certain argument positions in the family of semantic relationships around which a given discourse is structured; conversely, each of the arguments of a relationship (or function) appearing in a typical discourse is known to have some specific type, and the type of the output of each function which appears is known. In carrying a structure of this kind over to a programming language we may assume that basic set-theoretic relationships, and their connection with object types, are also understood, e.g. assume that a set s can be known to be "set-of-elements of type t". In addition to this, the basic facts that objects have attributes; that particular attributes of objects of a given type may themselves have known type; and that an object (which is really a vector) will frequently be defined when the value of all its attributes is known, can all be understood.

If available in a particular semantic environment, these facts can be used systematically (as indeed they are used in natural language discourse) to elide patterns of reference and to make them flexible. Specifically:

(i) If the arguments of a function or relation have known types, all of which are distinct, then when the function is invoked its arguments may be written in arbitrary order (since the correct order can be deduced from the known argument types.)

(ii) Generalising rule(i), suppose that we extend a device inherent in the 'expression' construct and agree that in each local context C of any program P, the last referenced object of each of the types declared relevant to P is implicitly available in C. Then a function argument can be omitted if its type is known and if the argument value desired is the implicit value of this type, in the sense just explained.

(iii) Generalising rule (ii), suppose that we keep the last k objects of each relevent type implicitly available. Then several function arguments of equal known types may be omitted if it is known that they must all be distinct, if Moreover, all these arguments play equivalent roles (so that they may be permuted), and if the k argument values desired are the last k objects of the relevant type, in the sense just explained. Moreover, dictions such as '<u>the former</u>', '<u>the first</u>', etc., can be provided, to distinguish between these implicit quantities when necessary. Finally, otherwise uninitialised variables of types deducible from syntactic context can be taken as references to the 'implicit' values of the type or types required.

Note that the convention just suggested serves as a partial replacement for the ordinary use of variable names in programming languages. From the point of view suggested by this convention, variable names can be understood simply as defining an indefinite variety of object types, each of which is the type of unique object in the program, namely the current value of the variable.

(iv) If an operation (e.g. a SETL primitive) admits arguments of a variety of types, we can specify the type of argument required at a given operator occurence by writing the name of the type in an argument position.

Note that, when taken together with rule (iii), this convention may suffice to define the arguments of an operator completely. For example, in a context in which two items x_1, x_2 of type *widget* are implicitly available, the notation eq *widget* clearly means 'x_1 eq x_2'.

(v) The functions and mappings known in a given program context will often return results and accept arguments of known type. The constraints which knowledge of this type implies can be used to allow one or more function arguments to stand for the function value required in a given context. We may even allow several functions to be involved, if the type relationships are sufficient to disambiguate the sequence of mappings which must be applied. Syntactically, we can allow a single argument to stand for itself, and use the notation $.(a_1, a_2, \ldots, a_n)$ when a few of the arguments, to one or more nested functions, are presented explicitly. For example, suppose a context in which an object of type t_1 is required, and where functions f and g, with arguments respectively of distinct types t_2, t_3 and t_4, t_5, and returning values of respective types t_1 and t_2 are available. Let x and y be variables of types t_2 and t_4 respectively. Then the notation $.(x,y)$ (or more specifically $t_1(x,y)$) can serve as an abbreviation for $f(x,g(y,z))$, where z is an 'implicit' value of type t_5, assumed to be available. We also allow an object with several declared attributes (see below) to be transformed automatically into some one if its attributes, provided that the particular attribute required is determined by the attribute's type. Another example is this: x can stand for <x> in a context in which a vector with components of a given type t is required, and where x is known to be of type t', or for the set {x} when a set of elements of type t is required.

(vi) If applied to a language L, rules (i-v) above impose a kind of 'case structure' on L, and allow abbreviations which exploit what can be thought of as principles of 'case agreement'. It may be worth extending this idea by using syntactic devices which are modeled even more directly after some of the case rules of natural language. For example, if a type name 'typn' is introduced, we can agree that variable name of the form 'typnx', 'typny', 'typnxy', 'xtypn' etc., will denote quantities of type 'typn'. Moreover, we can agree that 'typns' denotes the 'plural' form of 'typn', i.e., the type whose members are sets, all of whose elements are of one type 'typn'. For use with SETL, it is also useful to introduce a 'sequence case', i.e., to agree that 'typnseq' denotes the 'sequence form' of 'typn', i.e., the type of tuple objects all of whose components are of type 'typn'. We can also allow one variable of type 'typn', with no letters appended, to denote the currently implicit value of this type.

If a quantity 'typnx' of known type 'typn' is seen to be the value varied by a set-theoretic iterator, then it must vary over some range, and the type of its range must be of the plural type 'typns' or the sequence type 'typnseq'. If the set 'typnsy' over which the iteration is extended is implicit in the context of the iteration, we can allow $\forall typnx \in typnsy$: to be written. Much the same convention can be applied to iterations over the components of a sequence.

(vii) Rule (v) can be regarded as a kind of 'coercion rule', which calls for the replacement of one or more objects $o_1,\ldots,o_n$ of known kind by some other object o of a kind suiting the context in which the object o appears; and where o is obtained from $o_1,\ldots,o_n$ (and from still other objects if required) by application of available maps and functions. The maps and functions which can be used for this are in the first place those available as values or declared attributes of program objects having declared type; built-in, single-valued SETL operators having known action on object types may be used as well.

Among the SETL operators of which this may be said are {x}, which converts 'typn' to 'typns', <x>, which converts 'typn' to 'typnseq'; and #x, which converts both typn and typnseq to the built-in type integer. These last three constructions will be used only when no declared maps or functions producing an object of the type required by context from a supplied set of arguments is available, and when no object of the required type is implicitly available.

(viii) We can extend the coercion rules implicit in (v) and (vii) above by 'pluralising' it, i.e., by agreeing that if a sequence x of elements of type t' is called for, and if a map f coercing elements of type t into elements of type t' is available, then x may be coerced to $[+: y(n) \in x] <f(y)>$. Under corresponding circumstances, we can also allow coercion of sets of elements of type t to sets of elements of type t'.

(ix) A valuable principle, which disambiguates texts that would otherwise remain ambiguous, is the *principle of 'knitting'* which requires that all the elements specifically mentioned in a local or global context play some role in the text's disambiguated form. (This is a principle suggested by Jerry Hobbs in a study of the semantics of natural language.) A local variant of this principle requires that all the arguments supplied to a functional construct actually be used in the constructs disambiguated form; a global variant requires that, in deciding between two possible disambiguations of an entire subprocedure or program, we will prefer that which does not fail to reference any of the objects or object attributes declared with a program. In this same sense, a proposed disambiguation of a text is suspect if it implies that any computation is deliberately performed without its result ever being used.

(x) If a subfunction returns as value an object having as attributes several objects of various types, we can use an explicit or implicit call to the function solely for the value of some one of the attributes of the object it returns.

In this case, the other attributes of the object become implicitly available values of their several types. This remark applies in particular to iterators over objects 'typnseq' of sequence type. If such an iteration is written in its most abbreviated form, which is simply $\forall typn$ (rather than the fuller $\forall typn(k)$), the index k becomes an implicitly available value (of the system type integer).

(xi) If an object x of type t is declared to have attributes $at_1,\ldots,at_k$ (see Section 2 for the forms of declaration provided) then its several attributes can be referenced as $x.at_1,\ldots,x.at_k$. This notation can be used in either dexter or sinister position. As noted above (see(v)) we allow 'x' to be used instead of '$x.\ at_j$' when the context is dexter and a value of the type of $x.\ at_j$ rather than x is required in context. It is convenient to allow a similar, and indeed a more general, abbreviation on the left. Specifically, we allow

(1) $$x. = y$$

to abbreviate

(2) $$x.\ at_j = y$$

When y has a type suitable for assignment to the attribute $x.\ at_j$ of x (or when y may be coerced to a value suitable in this sense); provided, of course, that the resolution of (1) to (2) can be disambiguated. We also allow the multiple forms

(1') $$x. = y_1,\ y_2,\ldots,\ y_p$$

for

(2') $$x.at_{j_1} = y_1;$$

$$x.at_{j_2} = y_2;$$

$$\ldots$$

when the types of $y_1,\ldots,y_p$ are such as to dictate the individual assignments of (2') in a sufficiently unambiguous manner.

2. Specific syntactic conventions

It is now time to suggest specific syntactic conventions allowing the introduction of families of object types of the sort anticipated in the preceeding discussion. For this purpose, we introduce a number of declaration forms and linguistic conventions supplementing the existing rules of SETL. The first of these declaration forms is

(1) $tname_o$ <u>**has**</u> $aname_1$: $tname_1$, $aname_2$: $tname_2$,...,$aname_k$: $tname_n$;

Here, $tname_o$,...,$tname_n$ are type name (or somewhat more general constructs; see below). Moreover, $aname_1$,...,$aname_k$ are attribute names, which name attributes of objects of type $tname_o$.

The name $tname_o$ without parameters, can also function as an object name, and can be cast into the plural case and the sequence case. Objects x of type $tname_o$ are declared by (1) to have attributes $aname_j$ of respective types $tname_j$, and can be coerced by extraction of an attribute to an object of one of these types when context requires that this be done. In the presence of the declaration (1), the type name $tname_o$, supplied with appropriate parameters (some of which may be implicit) can function as an object-former for objects of this type. We also provide (1) in the modified form

(2) $tname_o$ <u>has</u> <u>identity</u>, $aname_1$: $tname_1$,...,$aname_k$: $tname_k$;

This declaration has a significance somewhat like that of (1), but the semantics of (2) differs in one essential regard from that of (1). Whereas an object of a type declared by (1) is essentially a SETL tuple, an object of a type declared by (2) is a SETL blank atom mapped onto a tuple by a behind-the-scenes system mapping *value*.

We allow declared attributes *attr* of an object x to be extracted by writing *x.attr*. In the case of an object x whose type is declared as (1), *x. attr* identifies a component of x; if the type of x is declared as (2), *x. attr* identifies a component of *value(x)*. The creation of an object x of type (1) merely involves the formation of a tuple t; the creation of an object x of type (2) involves both the formation of a tuple t and a call on the SETL operation <u>newat</u> to generate the value of x, with t then becoming *value(x)*. Similarly, insertion of x into a set means tuple insertion of x if of a type declared as (2).

In (1) and (2), $tname_j$ can be type names, but for $tname_j$ we also allow somewhat more general SETL-related type describing constructs, called *type descriptors*. These constructs can be built from type names *tname* in the following way: we can form

(3a) $\{tname\}$,

designating the type of a set whose elements are of type *tname*; and can form

(3b) $[tname]$

designating the type of a sequence whose elements are all of type *tname*. If $tn_1,\ldots,tn_k$ are a sequence of type names, then we also allow

(3c) $<tn_1,\ldots,tn_k>$,

which denotes the type of a tuple whose components are of types $tn_1,\ldots,tn_k$ respectively,

(3d) $[tn_1,\ldots,tn_k \rightarrow tname]$, denoting the type of a programmed function whose arguments are of types $tn_1,\ldots,tn_k$ respectively and whose result is of type *tname*, and

(3e) $$[\to tn_1,\ldots,tn_n]$$

denoting the type of a subroutine with arguments of the designated types.

These constructs can be compounded; e.g., we may write

(4) $$\{\langle tn_1,\ tn_2,\ \{tn_3\}\rangle\}$$

to describe a SETL map of two parameters with respective types tn_1, tn_2, which provides values which are sets of elements of types tn_3.

Note that $\{tn\}$ can also be written as *tns*, and $[tn]$ as *tnseq*.

If we wish to introduce a type which has only one attribute (whose name we need not distinguish from the type name), we can write

(5) *tname* is: *tdesc*;

where *tdesc* is a type descriptor of one of the forms (3).

We allow a new type name $tname_o$ to be declared by

(6) $tname_o$ <u>either</u> $tname_1,\ldots,tname_k$;

where $tname_1,\ldots,tname_k$ are type names or type descriptors. This states that an object of any of the types or descriptions $tname_j$ may be taken as an object of type $tname_o$ in a context requiring such an object. (i.e., coerced to an object of type $tname_o$ 'by generalisation'.)

3. A few examples.

It is now time to illustrate the conventions which we have proposed by applying these conventions to a number of examples. In our examples, we will also make use of the 'converge iterators' and related abbreviations described in SETL Newsletter 133B. As a first example, we shall give a code representing the basic LR(k)-parsing algorithm. In this code, we assume as usual that a finite-state automaton defined by a transition function *trans* is given, and that the states α of this automaton are arguments to a function *canfollow(α: string)*, where string is an (k+1)-tuple of symbols of the language being parsed. Written in a manner illustrating the conventions which have been described, this code is;

```
input is: tsymbseq;
prod has left:intsymb, right:intsmbseq;
symbol either intsymb, tsymb;
automaton has inistate: state, trans: {<state, symbol, state>};
canfollow  is: {<state, symbolseq, boolean>};
node either inode, tsymb;
inode has identity, kind: intsymb, descs: nodeseq;
definef lrparse (input, automaton, prods, canfollow);
nodeseq = input;
stateseq = inistate;
(∀)
    if ∃state(m), prod | nodeseq(m:n) is part eq prod and
         canfollow(left + nodeseq(m + n: k)) then
      nodeseq = nodeseq (1:m-1) + node(part) + nodeseq(m+n:);
      stateseq = stateseq(1:m);
    else if # stateseq is nss ℓt nodeseq then
         stateseq(nss + 1) = trans(nss,nss);
    end if;
end ∀;
return if   nodeseq eq 1 then nodeseq else Ω;
end lrparse;
```

This code, at first sight enigmatic or even erroneous, is justified by the following reflections:

i. (Line 3 of the subroutine.) Since *inistate* is a state, the line is corrected to stateseq = <inistate> by rule (v).

ii. (Line 5.) The range of variation of *state* is clearly *stateseq*, and of *prod* is clearly *prods*. The equality operator must compare objects of equal types, which must be obtained from objects of type *nodeseq* and *prod* respectively. Clearly then both must be coerced to *symbolseq*, *prod* by taking its *right*, and *nodeseq* by coercing each of its node components to *symbol*, which means using the *kind* field in the case of *inode* components. The uninitialised integer *n* is determined by length coercion. The field-name *left* is corrected to <prod, left> by the implicit reference rule and using the fact that context requires a *symbolseq*; and *nodeseq(m + n:k)* is also coerced (by application of the pluralised coercion rule) to an object of type *symbolseq*. The implicit argument of *canfollow* is clearly *state*. All in all, lines 5 and 6 of the preceeding code are corrected to

```
    if ∃state(m) ∈stateseq, prod ∈ prods |
      [n = # (prod.right); part = nodeseq(m:n);
     symbseq = [+: 1≤j≤n](if type (nodeseq(m+j-1) is nd) eq tsymb
                               then nd else nd. kind);
      return symbseq;] eq prod. right and
      canfollow(state,
    [+:1≤j≤k] if type (nodeseq(m+n+j-1) is nd eq tsymb  then
                                                  nd else nd. kind)
    then ...
```

iii. (Line 7.) *node(part)* is coerced, by the rule of implicit arguments and since a node when created must have a new identity, into <node(kind:prod.right,descs:part,newat)>. (Note that the context requires coercion of *node(...)* into *nodeseq)*.

iv. (Line 8) *nodeseq* must clearly be coerced into # nodeseq.

v. (Line 9) the first argument of trans must clearly be coerced into stateseq(nss), and the second argument into nodeseq(nss).

A few instructive observations concerning this example can be made. Most of the transformations which take the *lrparse* routine shown above into its SETL form are harmless from the point of view of efficiency. Applications of the pluralised coercion rule are exceptions; this rule introduces additional iterations whose subsequent removal by an optimiser may be difficult. In casting the process described by the above algorithm manually into an acceptable SETL form one will want to remove these troublesome iterations by applying strength reduction; i.e., one will choose to keep the string of symbols [+: nd(j) ∈ nodeseq] if <u>type</u> nd <u>eq</u> tsymbol then nd else nd. kind available. These remarks bring us easily to the following manually transposed form of our algorithm.

```
definef lrparse (input, automaton, prods, canfollow);
<inistate, trans> = automaton;
string = input; nodeseq = [+: c (n) ∈input] node(kind:c, descs:Ω, newat);
stateseq = <inistate>;
(∀)
     if ∃state(m) ∈ stateseq, prod ∈ prodns|
        string(m: #(prod.right)) is stringpart  and
          canfollow(state, <prod.left> + string(m+n:k)) then
        nodeseq = nodeseq(1:m-1)
               + node(kind:prod.left,descs:nodeseq(m+n:k), newat)
               + nodeseq(m+n:);
        stringseq = stringseq(1:m-1) + <prod.left> + stringseq(m+n:);
     else if # stateseq is nss lt # nodeseq then
          stateseq(nss + 1) = trans(stateseq(nss),stringseq(nss));
     end if;
end ∀;
return if # nodeseq eq 1 then nodeseq else Ω;
end lrparse;
```

Confirming a general observation made earlier, we see that the original form of our code is no shorter than its manually transposed form; but it is closer to a rubble of losely related fragments, and this is both psychologically more transparent and a more suitable target for some future automatic programming system.

Note that the second form of our algorithm can be optimised significantly by working with initial sequments of partseq and nodeseq, rather than with the whole of these vectors; this is the observation which eventually leads to an efficient code.

As a second example, we consider the Cocke-Younger-Earley 'nodal spans' parsing method. This is described by the following code-text:

```
gram has root: intsymb,syntypes: {<tsymb,intsymbs>}, prods:
                                    {<intsymb,intsymb,intsymbs>};
span has only start: integer, end: integer, kind: intsymb;
input is: tsymbseq;

definef nodeparse(inputseq, gram);
span = ::{span(end:n+1), ∀input,intsymb(input)} +
        {prodspan,∀spana,spanb.pfodspan| spana. end eq spanb.,start};
divlis = {<prodspan, spana,spanb>}, ∀ spana,spanb,prodspan};
     where prodspans = {span(spana.start,spanb.end,intsymb),
                                ∀intsymb(spana,spanb)}; end where;
if span(1,root,input + 1) not ∈ spans then return Ω;;
spans = ::{span} + [+: span,pair ∈ divlis{span}] pair;
return <spans, {<span, divlis{span}>,∀span}, ∃span |divlis{span} gt 2>;
end nodeparse;
```

Reduction of this text to standard SETL involves the following observations:

i. (Line 2 of the subroutine). $\forall$*input* becomes $\forall$*input(n)*$\in$ *inputseq*, and the two integers required for the *start* and *end* of the *span* formed in Line 2 are then identified with *m*. To obtain an *intsymb* with *input* as parameter, we must apply the map *syntypes*; thus *intsymb(input)* becomes *intsymb* $\in$ *symtypes(input)*. Moreover, $\forall$spana, spanb, prodspan becomes $\forall$spana $\in$ spans, spanb $\in$ spans, prospan $\in$ prospans. This same transformation occurs in Line 4.

ii. (Line 5 and 6). To form a set of objects of type *intsymb* from *spana* and *spanb* we extract the *kind* fields from both spans; since both *spana.kind* and *spanb.kind* must be knit into the set being formed, $\forall$*intsymb(spana,spanb)* is converted into $\forall$*intsymb* $\in$ *gram.prods.(spana.kind,spanb.kind)* Note that this relolution of the original intsymb(spana,spanb) is also supported by the principle of ''knitting'; if it is rejected, there will be no other program point at which *gram.prods* is used.

iii. (Line 8,9). The *span* implicitly referenced in {*span*} clearly *span(1,root,(#input) + 1)* (note that *input* + 1 is also converted into *#input* + 1 in Line 7). In the iterator which follows, [+: *span*,...] clearly abbreviates [+: span $\in$ spans,...]. This same transformation is applied to the $\forall$*span* and $\exists$*span* iterators which appear in Line 9.

4. Reflections on the foregoing.

Can a programming, language in order to reach a very high dictional level, reasonably allow free use of a system of implicit dictions like that described in the preceeding pages? For the following reasons, probably not. To attain significant compression of program text, one must skirt the border of ambiguity.

Therefore, logical mistranslations may result from the conversion of text containing implicit references to fully explicit SETL text. Consequently, the programmer who writes a text containing implicit references will generally have to check its transformed explicit version to assure himself that his logical intent has been correctly understood. Just as high a degree of skill will be required for this as for the manual generation of a fully explicit SETL text. Note however that text contining implicit references can in some cases give a better description of the underlying psychological process of program generation than fully explicit text will give, and 'implicit' text may therefore be preferable as a medium for initial program specification, and also for explanation of algorithms, especially since the system of implicit reference we have proposed resemebles that employed in natural language. Since certain types of common errors should be catchable by cross-checking two texts, one of which is machine-generated, and where both texts are supposed to represent the same process, it may also be valuable to generate an explicit SETL text mechanically from an implicit SETL text, and then to work with and debug the explicit rather than the original text. Another approach,which may be more practical,is to work always with fully explicit text, but to check for errors using a type analysis like that needed to resolve implicit references.

The preceeding remarks suggest that an interactive 'semi-automatic programming' system might be structured as follows. As source text, it could admit programs like that described in the preceeding paragraphs. Type declarations could be entered first, followed by the imperative parts of a program text. As each small group of imperative statements was entered, the system could emit a series of yes/no questions intended to confirm the manner in which the system intended to resolve implicit references.

At each logical point where such a resolution was required, a reasonably small number, say a half-dozen or a dozen possible resolutions, might be generated internally, in order of diminishing plausibility. If what the system took to be the most plausible resolution was interactively rejected by the user, these other resolutions might be displayed for his choice. Finally, overall consistency checks, such as the principle of 'knitting', could be applied.

For the number of mistaken interpretations generated during the resolution of implicit references to be kept minimal, and for confidence in the overall result of such a transformation to be justified, formal principles which somewhat 'overdetermine' the transformation may have to be found. Superficial, essentially linguistic principles like those sketched in Sections 1 and 2 may be insufficient, in which case we may need rules which rest on a deeper analysis of the mathematical structure of a program and on some inkling of the informal correctness proof which underlies it. Information of this depth is not easy to come by, and if it turns out of be necessary to penetrate to this depth the development of highly automatic versions of the techniques which have been suggested in the preceeding pages may slow down to match the development of automatic proof techniques and of automatic techniques for analysing the correctness of programs. Note however that by adding assertions to be checked at run-time to a program text containing implicit references, we can increase its redundancy significantly, and have considerably more confidence in-the explicit text which results from it by transformation.

Fully automatic programming systems intended to work from natural language source text face heavier sledding yet. They must first analyse their natural language input and transform it sucessfully into a collection of declarative and imperative formal statements like those envisaged in

Section 1 and 2 above. They must be able to transpose requests for confirmation of an interpretation into acceptable natural language output forms. They must probably be able to prepare an overall natural-language document summarising the information gathered in

5. A few remarks on verifying assertions.

As noted in Newsletter 135A, loops in programs will, when they are not driven by the repetitive structure of some intenal or external data object, often arise from the transformation into fixed-point form of an underlying mathematical specification which in its pure form refers to a set S too vast to be searched explicitly. That is, within the (very large) set S defined by a predicate C, the loop constructs an element x having some defining property C_1 by using an initial element $x_1 \in S$ and a transformation ϕ such that the iteration $x_{n+1} = \phi(x_n)$ eventually leads from x_o to the desired x. Suppose tht such an iteration has been programmed, and has produced an x. We can often test the correctness of our program by checking that x has both properties C and C_1. If evaluation of the predicate C_1 is impossibly expensive, for example if C_1 asserts that x has some extremal property, we can in place of $C_1(x)$ check some mathematically equivalent property of x. Suppose that, to allow demands for verification to be inserted into a code, we introduce an assert statement of the form

(1) assert C;

where C is some boolean valued expression. If such a statement can be seen to be true by static program analysis, our program will have been verified mathematically. Even where this is infeasible, we may imagine (1) to be checked dynamically; a checking operation of this sort can be regarded as a substitute for some of the manual examination of intermediate data that would otherwise have to be performed during program debugging. Since it is the predicates C and C_1 which define the purpose of an algorithm, verifying assertions are generally not too hard to formulate (if assertions are required only for dynamic checking, and not for static verification.) For example, the point of the *lrparse* code shown in section 3 is to form a parse tree; thus we can check the correctness of this code by declaring that

```
tnodsunder is:{<node, symbolseq>}
```

and by interpolating the following calculations immediately prior to the penultimate line of the code:

```
if nodeseq eq 1 then   /* check will be performed */
     nodes = :: nodeseq + [+: node]→descs;
     assert  ∀node | if type eq inode then prod(node,descs) ∈ prods;
     tnodsunder = :: {<node, kind>, ∀node | type node eq tsymb}
          + {<node, [+: desc] tnodsunder(desc)>
               | tnodsunder(node) eq Ω and ∀desc|tnodsunder(desc)ne Ω};
     assert tnodsunder (nodeseq) eq input;
end if;
```

Note however that a rather more sophisticated assertion is required if we mean to check that the code also performs properly in cases when Ω is returned.

The nodal spans algorithm given toward the end of section 3 may be checked in a rather similar way.