ABSTRACTION and VERIFICATION in ALPHARD: A Symbol Table Example

Ralph L. London, USC Information Sciences Institute

Mary Shaw, Carnegie-Mellon University

Wm. A. Wulf, Carnegie-Mellon University

December 29, 1976

Abstract: The design of the Alphard programming language has been strongly influenced by ideas from the areas of programming methodology and formal program verification. In this paper we design, implement, and verify a general symbol table mechanism. This example is rich enough to allow us to illustrate the use as well as the definition of programmer-defined abstractions. The verification illustrates the power of the form to simplify proofs by providing strong specifications of such abstractions.

Keywords and Phrases: abstraction and representation, abstract data types, assertions, correctness, data abstraction, data structures, information hiding, modular decomposition, program specifications, program verification, programming methodology, structured programming

The research described here was supported in part by the National Science Foundation (Grant DCR74-04187) and in part by the Defense Advanced Research Projects Agency (Contracts: F44620-73-C-0074, monitored by the Air Force Office of Scientific Research, and DAHC-15-72-C-0308). The views expressed are those of the authors. This report is one in a series being printed jointly by CMU and ISI.

Contents

Introduction--The Symbol Table Task

Previous reports [Shaw76b, Wulf76a,b] have described the Alphard programming language and its associated verification methodology. These reports developed Alphard definitions for the canonical examples of data abstractions (stacks, queues, and sets). These examples are sufficiently simple to be grasped readily, and they have appeared often enough in other languages that the reader may compare various approaches to their definition. There is, however, a danger in considering *only* these examples. It is possible that an approach will work for only the easy examples, or that the definition of something more complex will be far less elegant.

Therefore, in this report we shall consider a larger, more realistic example: an abstraction of a *symbol table*. For comparison purposes the reader may wish to refer to the similar example given in [Guttag76] and to a hashtable example in [Wegbreit76].

Suppose that we must produce a number of compilers, assemblers, and interpreters to operate on several different computers. Each such system will contain a symbol table mechanism; although each system will have its own requirements, many of the gross, *abstract* properties of these symbol tables will be the same. It seems desirable to have a single implementation of these common aspects which is verified; that will be our aim.

But what *are* the common properties? Many texts [e.g., Gries71] describe a symbol table as a *mapping* from identifiers (strings appearing in a source program) to a set of *attributes* associated with those identifiers. Examples of such attributes include "type", "run-time memory address", "number of dimensions" (for arrays), etc. In some cases, the mapping may be sensitive to the context in which the identifier occurs. (Algol-like block structure is the most common example of this context sensitivity; the mapping from identifier to attributes depends upon the block in which the identifier appears. Name qualification, as in field selection from a record, is another example in which the interpretation of the field selector depends upon the type of the record.) The common properties, then, are ones which involve the application and manipulation of this mapping; principally

- some means to apply the mapping, i.e., to find the attributes associated with the occurrence of an identifier.

- some means to alter the mapping, e.g., by inserting and/or deleting entries and signaling changes in context.

Since we want our abstraction to serve a spectrum of languages, system types (e.g., compilers and assemblers), and machines, it would *not* be appropriate to include the specific attributes as part of the abstraction. Rather, we shall presume that the user of our abstraction will define some mechanism for storing and retrieving attributes, e.g., a vector of records; our abstraction will then provide a mapping from an identifier to a unique integer which, for example, may then be used as an index into this vector of attribute records.

Concerning the issue of context sensitivity, we shall provide an abstraction which supports block structure because (1) it is the more general case and (2) with proper implementation, the generality costs very little when it is not used. We shall not explicitly provide for the kind of context sensitivity needed for record selectors, but we shall show how the abstraction may be used to achieve it.

Note that the informal term "block-structured" does not describe a unique name-binding policy. For example, consider the program fragment

```
. . .
integer k=10;
. . .
    begin
    vector X[1:k];
    integer k=3;
    . . .
```

In the declaration of the vector "X", there is a question about which "k" should be used to define its upper bound. The semantics of some languages specify that the value of the variable "k" defined at the outer block level, i.e., 10, should be used; other languages specify that it is the innermost definition, i.e., "integer k=3", which should be used. To accommodate the second of these schemes requires that a full lexical analysis pass be performed before *any* name binding (symbol table construction) is done.

In order to make our abstraction useful on this pure lexical pass, as well as later when the full symbol table is constructed, we shall define it as a mapping between "things" and integers. In a simple system the "things" will be identifiers and the integers will probably be indices into the vector of attributes described above. In a more complex system, the initial lexical pass may use the abstraction to convert identifiers into integers; these integers may in turn be the "things" mapped into symbol table indices during a later pass. An example of the use of the abstraction will be given later to help clarify this point; for the moment the reader may simply assume that the "things" are identifiers.

Summarizing, then, our abstraction shall provide:

(a) A block-structured mapping from "things" to integers.

(b) A set of six operations to *insert* a new "thing", to *lookup* the integer associated with a specific "thing", to test whether a specific "thing" is *defined* at the current block level, to *enter* and to *leave* a block level, and to test whether the mapping is *full*, i.e., whether there is room for another "thing".

The Symbol Table Abstraction

The preceding section provides an informal description of the symbol table abstraction; in this section we shall be more precise. Specifically, the specifications part of the form called "symtab" is:[1]

```
form symtab(T:form< ←,=,hash(T,k:integer) returns x:integer pre (k>0) post (0≤x<k') >,
        m,n:integer) =
  beginform
  specifications
      requires n≥1 ∧ m≥1;
      let symtab = <block:integer, assoc:{<s:T,bl:integer,ui:integer>}>;
      invariant
          cardinality(assoc)≤n
          ∧ 1≤ui≤n ∧ 1≤bl≤block
          ∧ ($t_1$,$t_2$ ∈ assoc ⊃ ($t_1$.s=$t_2$.s ∧ $t_1$.bl=$t_2$.bl ≡ $t_1$.ui=$t_2$.ui));
      initially symtab = <1,{}>;
      functions
          defined(st:symtab,str:T) returns t:boolean
              post t = ∃i st <str,st.block',i> ∈ st.assoc,
          insert(st:symtab,str:T) returns i:integer
              pre cardinality(st.assoc) < n ∧ ¬defined(st,str)
              post  st = <st.block', st.assoc' ∪ {<str,st.block',i>}>,
          lookup(st:symtab,str:T) returns x:integer
              post if ∃ y ∈ st.assoc st [y.s=str ∧ ∀z ∈ st.assoc, z.s=str ⊃ z.bl ≤ y.bl]
                  then x = y.ui
                  else x = 0,
          enterblock(st:symtab)
              post st = <st.block'+1,st.assoc'>,
          leaveblock(st:symtab)
              pre st.block > 1
              post st = <st.block'-1, st.assoc' - {<s,x,ui> st x≥st.block'}>,
```

[1] A primed variable (e.g., k') represents the value of that variable prior to the execution of an operation. To shorten the pre, post, in, and out conditions in our papers, we often, by convention, omit assertions about variables which are completely unchanged. Thus for example, we have omitted st = st' from the post condition of *defined.*

```
full(st:symtab) returns t:boolean
    post t = (cardinality(st.assoc) = n);
```

Note that, abstractly, a symbol table consists of a pair: an integer, "block", and a set, "assoc". The integer denotes the current block level, has the initial value 1, and is altered only by the operations *enterblock* and *leaveblock*. The set, initially empty, consists of triples containing the "thing" defined, the block level at which it was defined, and the unique integer ("ui") associated with the <thing,block level> pair.

The parameters of the form specify the type, "T" (usually strings), of "things" to be entered in the table, and the maximum number, "n", of simultaneous entries permitted. The parameter "m" is a bit more difficult to explain, and we shall for a moment defer it, together with the discussion of the required rights of T.

Since the symbol table contains only currently defined things, the block level of each entry must be legitimate (e.g., between 1 and the current value of "block"). Further, since a maximum of n entries is allowed, the "associated integer" must lie between 1 and n. The let clause and the abstract invariant express these restrictions (the last line of the invariant expresses the uniqueness of the integer associations). The remainder of the specifications states that the initial symbol table has a block level of 1 and an empty "assoc" set, and then lists the symbol table functions and their abstract pre and post conditions.

Now, let us return to the issue of the parameter m and the required rights on T. As may be seen from the requires clause of the specifications, the only requirement on m is that its value be strictly positive; it does not enter into any of the other parts of the formal specification. Hence, one may properly conclude that its precise value is immaterial and the abstraction will function correctly with any positive value.

The value of m does, however, affect the *performance* of the abstraction. Neither Alphard nor other languages with similar goals have yet found an appropriate way to specify performance properties. In practical systems, of course, such properties are of paramount importance. Since we now have no formal way of specifying them, we must give a small peek into the representation in order to explain the significance of m. (Indeed, the need to have m and the hash function name in the specifications has essentially revealed the techniques used in the implementation of the abstraction.) The representation uses a hash table, with collisions resolved by chaining, and m specifies the size of this table, i.e., the number of values that the hash function may assume. Although any positive value of m will work, larger values will tend to provide faster searches at the expense of some additional storage.

In addition, the value of m may affect the distribution of "hits" on any particular hash table entry; see [Knuth73] for a discussion of hashing functions and their properties. We will not discuss these properties here, but note that the form T which defines the things stored in the symbol table is required to provide a hashing function which, given an object of type T and an integer k, returns an integer in the range 0 to k-1. Thus, an appropriate choice of m depends in part on the properties of this function.

Implementation of Symbol Table

In choosing the implementation of the symbol table abstraction, we have been careful to pick a practical one; it is, in fact, one which is used in several commercial compilers. We chose to do this rather than, for example, to use a direct implementation in terms of sets (e.g., the *simpleset* form defined in [Shaw76b]). We have done this in order to emphasize that both the language and verification methodology are intended to be used for practical, production quality systems. The more direct implementation, and also its proof, would have been straightforward and clear. However, it would not have been a production quality implementation and thus would not have been useful in a real system. We shall comment on this point further in the conclusion, but we feel strongly that language, methodology, and verification *must* respond to the requirements of practical, efficient systems.

We shall obtain the implementation in two steps. We shall define an intermediate abstraction (form) in the process of obtaining the complete implementation. This intermediate abstraction will support a restricted, but not uncommon, style of list-processing.

Now, whenever a system implementation is described, one is faced with a presentation problem: whether the description should be "top-down" or "bottom-up". Both have advantages. In this case we have chosen to make the presentation predominantly top-down -- primarily to emphasize that the implementation of lower level abstractions is irrelevant to the correctness of the higher level ones. The next paragraph, however, is an exception to the predominant flavor of the presentation; it describes the implementation of the symbol table in low-level terms, as it will exist after compilation of the *forms*. It is included for those of us (including the authors) who still need concrete representations to aid their reasoning; purists may simply skip the next paragraph.

The symbol table will be implemented as a hash table with explicit entries for the symbol and its declaration block level, but an implicit encoding of the integer mapping. Hash collisions are resolved by associating a linked list of symbol table entries with each value of the hash function. Each new entry is inserted at the head of the appropriate list. The entries on the lists are therefore ordered by block level (innermost block first). To find the innermost instance of a symbol, *lookup* need only perform a linear search of the list associated with the hash value of the symbol; the first instance of the symbol in the list is necessarily the one declared at the innermost block level. It is a simple matter for *leaveblock* to delete the proper entries from the heads of these lists.

The implementation of symtab presumes the existence of a form called "condis" (collection of named, disjoint integer sequences). The explanation of the symtab implementation will require that we first understand (i.e., specify) condis. Although condis is intended to support a group of linear lists, its abstract specification is stated in terms of more mathematically tractable entities, namely sets and sequences.[2] The verification of symtab will use the abstract specification from condis but nothing else. The verification of condis will be independent of symtab and its verification. The specifications part of condis is:

```
form condis(n,m:integer) =
beginform
specifications
  requires $n \geq 1 \wedge m \geq 1$;
  let condis = $L:\{sq_i:<e_{i1},e_{i2}, \ldots , e_{in_i}> \mid 0 \leq i \leq m-1 \wedge e_{ik}$ is integer$\}$;
  invariant $1 \leq e_{ik} \leq n \wedge \forall i,j \in [0..m-1](e_{ik_1}=e_{jk_2} \supset i=j \wedge k_1=k_2)$;
  initially $\forall i \in [0..m-1]\ sq_i = <>$;
  functions
     xtnd(s:condis,i:integer) returns j:integer
        pre $i \in [0..m-1] \wedge \mathrm{SIGMA}_{j \in [0..m-1]} \mathrm{length}(s.sq_j)<n$,
        post $s.sq_i = <j> \sim s.sq_i'$,          ! note j is a new value not in any sq (by $I_a$)
     del(s:condis, i,j:integer)
        pre $s.sq_i = < \ldots , j, \ldots > \wedge i \in [0..m-1]$
        post $s.sq_i = <j, \ldots >$,
     delall(s:condis,i:integer)
        pre $i \in [0..m-1]$
        post $s.sq_i = <>$,
     full(s:condis) returns t:boolean
        post $t = \mathrm{SIGMA}_{j \in [0..m-1]} \mathrm{length}(s.sq_j) = n$;
 generator indis(s:condis,i:integer) extends x:integer
     requires $0 \leq i \leq m-1$
     let indis = $s.sq_i$ where indis$\neq$<> $\supset$
           (indis = c~<x>~d and c, <x>, and d are disjoint);
     rule for(I, x, <s,i>, ST) =
        premise $s.sq_i$=c~<x>~d $\wedge$ I(c) {ST} I(c~<x>);
     rule first(P, x, <s,i>, $\beta$, $S_1$, $S_2$, Q) =
        premise $s.sq_i$=c~<x>~d $\wedge P \wedge \forall y \in c(\neg\beta(y)) \wedge \beta(x)$ {$S_1$} Q,
        premise $P \wedge \forall y \in s.sq_i \neg\beta(y)$ {$S_2$} Q;
 auxiliary predicates
     follows(s:condis,i,j:integer) $\equiv_{df} \exists k$ st $sq_k = < \ldots, i, \ldots, j, \ldots >$,
     mbr(s:condis,i,j:integer) $\equiv_{df} sq_i = < \ldots, j, \ldots >$;
```

A condis is abstractly described as a set of precisely m sequences of integers; these

[2] Definitions and properties of sets appear in [Halmos60] and those of sequences in [Wulf76a,b].

sequences are named sq_0 through sq_{m-1}. The abstract invariant asserts that: (1) each integer in any of the sequences lies in the range 1 to n and (2) a particular integer appears as a sequence element at most once in the entire set of sequences. From these two facts we can observe that the sum of the lengths of the sequences is at most n; moreover, in the case that this sum is n, each of the integers 1 through n will appear (precisely once) in one of the sequences.

As a practical matter, each of the sequences in the condis will represent a linear list; specifically, sq_i will be associated with the value i produced by the hash function. The sequence elements will be the (integer) indices into a vector of information within symtab; thus the sequence sq_i (and the corresponding entries in the vector of information) will represent the linear list of triples in the abstract "assoc" set of symtab which have the hash function value i.

Four functions and a generator are provided by the condis form. Function ***xtnd*** extends the head of a specified sequence by one element; the abstract invariant prevents this integer from being one which already appears in some sequence. Function ***del*** permits the initial elements of a specified sequence to be deleted, and function ***delall*** permits all the elements of a specified sequence to be deleted. Function ***full*** tests whether all of the integers already are in some sequence. Generator ***indis*** produces the elements of a specified sequence in order, starting with the head. The specification of condis also gives two auxiliary predicates (***follows*** and ***mbr***). These may be used in proofs, but are not actually implemented as executable functions; they should be viewed as an extension to the abstract vocabulary.

At first sight, the condis abstraction may seem unusual; however, we chose to define it in this way for two reasons:

- By using integers to denote elements, we can obtain an efficient encoding of the unique integer mapping required by symtab. This encoding is one which might be selected in actual practice.

- This definition allows us to skirt the issue of pointers (references) for purposes of this paper.[3]

Now we can present the complete definition of the symtab form.

[3] As most people who have followed the recent literature on programming methodology and verification are aware, the presence of references (unconstrained pointers) in a programming language interferes with our ability to understand and verify programs that use them. While we believe we have made significant progress in Alphard toward resolving the problems introduced by the unconstrained pointer, we will not complicate this paper with pointer issues.

```
form symtab(T:form< ←,=,hash(T,k:integer) returns x:integer pre (k>0) post (0≤x<k') >,
      m,n:integer) =
  beginform
  specifications
     requires n≥1 ∧ m≥1;
     let symtab = <block:integer, assoc:{<s:T,bl:integer,ui:integer>}>;
     invariant
        cardinality(assoc)≤n
        ∧ 1≤ui≤n ∧ 1≤bl≤block
        ∧ ($t_1$,$t_2$ ∈ assoc ⊃ ($t_1$.s=$t_2$.s ∧ $t_1$.bl=$t_2$.bl ≡ $t_1$.ui=$t_2$.ui));
     initially symtab = <1,{}>;
     functions
        defined(st:symtab,str:T) returns t:boolean
           post t = ∃i st <str,st.block',i> ∈ st.assoc,
        insert(st:symtab,str:T) returns i:integer
           pre cardinality(st.assoc) < n ∧ ¬defined(st,str)
           post  st = <st.block', st.assoc' ∪ {<str,st.block',i>}>,
        lookup(st:symtab,str:T) returns x:integer
           post if ∃ y ∈ st.assoc st [y.s=str ∧ ∀z ∈ st.assoc, z.s=str ⊃ z.bl ≤ y.bl]
              then x = y.ui
              else x = 0,
        enterblock(st:symtab)
           post st = <st.block'+1,st.assoc'>,
        leaveblock(st:symtab)
           pre st.block > 1
           post st = <st.block'-1, st.assoc' - {<s,x,ui> st x≥st.block'}>,
        full(st:symtab) returns t:boolean
           post t = (cardinality(st.assoc) = n);

representation
  unique
     blvl: integer,
     info: vector(record(s:T,bl:integer),1,n),
     as: condis(n,m)
        init blvl ← 1;
  rep(as,info,blvl) = <blvl, {<info[i].s,info[i].bl,i> | ∃j ∈ [0..m-1] st mbr(as,j,i)}>;
  invariant
     (mbr(as,i,j) ⊃ hash(info[j].s,m) = i)
     ∧ (follows(as,i,j) ⊃ blvl ≥ info[i].bl ≥ info[j].bl ≥ 1 ∧ (info[i]=info[j] ⊃ i=j))

implementation
  body defined out (t = ∃j st st.info[j]=<str,st.blvl> ∧ mbr(st.as,hash(str,m),j)) =
     first j:indis(st.as,hash(str,m)) suchthat st.info[j].s=str
        then t ← st.info[j].bl=st.blvl else t ← false;
```

```
    body insert in ¬full(st.as) ∧ ¬defined(st,str)
            out (st.info[i]=<str,st.blvl> ∧ sq_hash(str,m) = <i>~sq'_hash(str,m)) =
        begin
        i ← xtnd(st.as,hash(str,m));
        st.info[i] ← <str,st.blvl>;
        end;

    body lookup out (x=0 ⊃ (j ∈ [1..n] ∧ ∃i ∈ [0..m-1](mbr(st.as,i,j)) ⊃ st.info[j].s ≠ str)) ∧
            (x>0 ⊃ st.info[x].s=str ∧ (st.info[j].s=str ⊃ j=x ∨ st.info[x].bl > st.info[j].bl)) =
        first j:indis(st.as,hash(str,m)) suchthat st.info[j].s=str
            then x ← j else x ← 0;

    body enterblock out (st.blvl = st.blvl' + 1) =
        st.blvl ← st.blvl+1;

    body leaveblock in st.blvl > 1
            out (st.blvl = st.blvl' - 1 ∧ (j ∈ [1..n] ∧ i ∈ [0..m-1] ⊃
                (mbr(st.as,i,j) ≡ mbr(st.as',i,j) ∧ st.info[j].bl < st.blvl'))) =
        begin
        st.blvl ← st.blvl-1;
        for i: upto(0,m-1) do              ! the generator upto is defined in [Shaw76b]
            first j:indis(st.as,i) suchthat st.info[j].bl ≤ st.blvl
                then del(st.as,i,j) else delall(st.as,i);
        end;

    body full out (t = SIGMA_j∈[0..m-1] length(s.sq_j) = n) =
        t ← full(st.as);
endform
```

Note that the representation of a symtab consists of three objects: (1) *blvl*, an integer, is a direct representation of the abstract entity *block*, and is initialized to 1. (2) *info* is a vector of records which hold the "thing" (usually a string) and the block level at which it was declared. Each of these records is, in effect, one of the triples in the abstract "assoc" set; the third element of the triple, the unique integer, is not explicitly represented -- rather, it is implicitly encoded as the index of this record in the vector. (3) *as* is a condis, and as explained above, it represents a set of lists of indices into this vector of records; each such list is uniquely associated with a hash function value.

A point which may not be obvious is worth noting. It is rare that all *info* entries will be in use; we thus have a potential problem in maintaining the free storage of this vector. This problem is handled by the condis abstraction. The uniqueness of the integers in condis sequences guarantees that no *info* entry will be used simultaneously by different members of assoc. In essence, the integer values which are in the condis sequences correspond to occupied entries, and all other integers in the range 1 to n correspond to unoccupied, or free, entries. Specifically, the abstract invariant of condis and the post condition of *xtnd* together provide a safe allocation of new *info* entries. Similarly, *del* and *delall* provide a safe deallocation mechanism.

To illustrate the operation of the implementation, consider the interaction of the bodies of *insert* and *lookup*. When a new symbol is to be inserted, we first invoke the condis operation *xtnd*. This has the effect of extending the head of the sequence associated with the hash value of the symbol by a new, unique, integer. This integer is then used as the index into the vector *info* and the symbol and current block level are recorded in this entry. When a later *lookup* is performed on this symbol, the *indis* generator is used to find the first integer, j, in the sequence associated with the hash value of the symbol for which "info[j].s" matches. Since *xtnd* extends the sequence at its head, this match is necessarily the most recently declared instance of the symbol.

Verification of the form Symtab

A form is verified by proving four properties as described in [Wulf76a,b] and summarized in Appendix A. As promised earlier, the verification below uses only the abstract specification of the form condis, including the auxiliary predicates. The implementation of condis is, as desired, irrelevant to symtab. All uses of the generator indis satisfy the independence assumption provided that in leaveblock we regard both the then and else clauses as being outside the first generator.[4]

For the form

1. Representation validity
   Show: I_c(as,info,blvl) ⊃ I_a(rep(as,info,blvl))
   Proof: cardinality(assoc) $\le$ n follows from I_a for condis, namely, $1 \le e_{ik} \le n$ and no duplicate e_{ik}'s means at most n elements in assoc. The relation $1 \le ui \le n$ holds because of mbr in the rep function and $1 \le e_{ik} \le n$ in I_a for condis. The relation $1 \le bl \le block$ follows by setting j=i in follows(as,i,j) in I_c. To show uniqueness in assoc, first note that identical s and

[4] Strictly speaking, this violates the definition of the first statement in [Shaw76b], a definition which we must modify to permit, for example, finalization statements and the leaveblock usage. We must also weaken the independence assumption. With the strict interpretation, however, an ad hoc argument shows that there are no problems in this case because indis does not modify the generated sequence and no further generation is attempted after the then and else clauses.

identical bl means, letting hash(t1.s,m) = hash(t2.s,m) = k, that mbr(as,k,t1.ui) and mbr(as,k,t2.ui), whence we have either follows(as,t1.ui,t2.ui) or follows(as,t2.ui,t1.ui). In either case, since info[t1.ui]=info[t2.ui], then t1.ui = t2.ui as required. The converse of the uniqueness clause holds since I_a for condis means no duplicates.

2. Initialization

   Show: $n\geq 1 \wedge m\geq 1$ { blvl←1 } <1,{}> = rep(as,info,blvl) $\wedge\ I_c$

   Proof: This holds since initially of condis says each sq_i=<>, i.e., ¬mbr(as,j,i) and ¬follows(as,i,j). Note that $n\geq 1 \wedge m\geq 1$ permits the declaration as:condis.

For the function defined

3. Concrete operation

   Show: I_c { first j:indis(st.as,hash(str,m)) suchthat st.info[j].s=str then t ← st.info[j].bl=st.blvl else t←false } $\beta_{out} \wedge I_c$

   Proof: I_c holds since it is unchanged. Indis may be called since 0≤hash(str,m)<m. By the first term of I_c, str can only be located from $sq_{hash(str,m)}$. For the then clause, the second term of I_c gives β_{out}. (Note that mbr(st.as,hash(str,m),j) holds by the definition of indis.) For the else clause str was not located from $sq_{hash(str,m)}$, whence t is false as required.

4a. β_{in} holds

   β_{in} is true

4b. β_{post} holds

   Show: $I_c \wedge \beta_{out} \supset t = \exists i$ *st* <str,st.block',i> ∈ st.assoc

   Proof: If t is true in β_{out}, then <st.info[j].s, st.info(j).bl,j> = <str,st.block',i> ∈ st.assoc, i.e., choose i to be j. If t is false in β_{out}, there will be no i and t is false as required.

For the function insert

3. Concrete operation

   Show: $\beta_{in} \wedge I_c$ { i←xtnd(st.as,hash(str,m)); st.info[i]←<str,st.blvl> } $\beta_{out} \wedge I_c$

   Proof: The pre of xtnd holds because hash(str,m) ∈ [0..m-1] and because ¬full(st.as) means cardinality(st.assoc) < n whence the SIGMA term < n. The first term of β_{out} is clear. Since the hash(str,m)th sequence of as is extended, $sq_{hash(str,m)}$ = <i>~$sq'_{hash(str,m)}$ where i is the appended new element. The first term of I_c follows by the call to xtnd and st.info[i].s=str; the second term of I_c follows by I_c and ¬defined(st,str), i.e., str is not defined at the current block.

4a. β_{in} holds

   Show: I_c ∧ cardinality(st.assoc)<n ∧ ¬defined(st,str) ⊃ β_{in}

   Proof: cardinality(st.assoc) < n means ¬full(st.as).

4b. β_{post} holds

Show: $I_c \wedge \beta_{pre} \wedge \beta_{out} \supset \beta_{post}$

Proof: The new triple <st.info[i].s,st.info[i].bl,i> is added to st.assoc.

For the function lookup

3. Concrete operation

Show: I_c { first j:indis(st.as,hash(str,m)) suchthat st.info[j].s=str then x←j else x←0 } $\beta_{out} \wedge I_c$

Proof: I_c is unchanged. As in the operation defined, str can only be located from $sq_{hash(str,m)}$. By indis, j ∈ [1..n]. Hence only the else clause makes x=0 and, as required in this case, j ∈ [1..n] ∧ ∃i ∈ [0..m-1](mbr(st.as,i,j)) ⊃ st.info[j].s≠str. For the then clause, the first term after x>0 holds by the suchthat clause. For the second term after x>0, suppose j≠x. Using the second term of I_c (note that follows(st.as,x,j) holds) rules out the possibility that st.info[x].bl=st.info[j].bl since otherwise j=x. Hence st.info[x].bl > st.info[j].bl.

4a. β_{in} holds

β_{in} is true

4b. β_{post} holds

Show: $I_c \wedge \beta_{out} \supset \beta_{post}$

Proof: x=0 means ¬∃y *st* y.s=str. x>0 means x = y.ui, i.e., y = <st.info[j].s,st.info[j].bl,j>.

For the function enterblock

3. Concrete operation

Show: I_c { st.blvl ← st.blvl+1 } $\beta_{out} \wedge I_c$

Proof: β_{out} is clear. Since st.blvl increases, I_c still holds.

4a. β_{in} holds

β_{in} is true

4b. β_{post} holds

Show: $I_c \wedge \beta_{out} \supset \beta_{post}$

Proof: st.block = st.blvl = st.blvl'+1 = st.block'+1 and st.assoc = st.assoc'.

For the function leaveblock

3. Concrete operation

Show: $\beta_{in} \wedge I_c$ { body } $\beta_{out} \wedge I_c$

Proof: st.blvl = st.blvl'-1 is clear. By the for statement each sq_i for i ∈ [0..m-1] is adjusted by the first statement. For each of indis, del, and delall, we have the pre condition i ∈ [0..m-1] by the for statement. The other part of pre of del, mbr(st.as,i,j), holds by indis. In the then case, del(st.as,i,j) deletes all entries in sq_i up to but not including j. Because j is the first j with st.info[j].bl≤st.blvl<st.blvl', the block level ordering asserted by I_c ensures β_{out}. In the else case all st.info[j].bl>st.blvl whence sq_i should become <>, which delall does. β_{out} follows since st.info[j].bl<st.blvl' ≡ ¬mbr(st.as,i,j). In both the then and else cases, I_c still holds because the lists only get shorter and st.blvl>1 on entry.

4a. β_{in} holds
Show: $I_c \wedge$ st.block >1 ⊃ st.blvl>1
Proof: In the rep function, st.block and st.blvl correspond.

4b. β_{post} holds
Show: $I_c \wedge \beta_{pre} \wedge \beta_{out} \supset \beta_{post}$
Proof: Since st.blvl=st.blvl'-1, we have st.block=st.block'-1 as required. By β_{out} and the rep function, st.assoc=st.assoc' - {<s,x,ui> *st* x≥st.block'}.

For the function full

3. Concrete operation
Show: I_c { t←full(st.as) } $\beta_{out} \wedge I_c$
Proof: β_{out} is exactly the post condition of full in condis. I_c is unchanged.

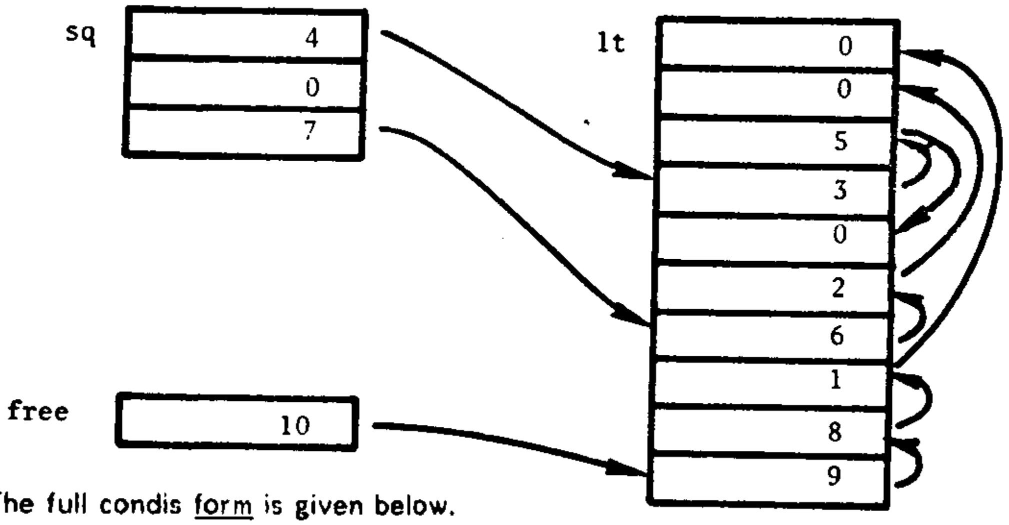
4a. β_{in} holds
β_{in} is true

4b. β_{post} holds
Show: $I_c \wedge \beta_{out} \supset \beta_{post}$
Proof: t = (SIGMA$_{j \in [0..m-1]}$ length(s.sq_j) = n) = (cardinality(st.assoc) = n).
QED

Implementation of the form Condis

As discussed earlier, the abstract representation of condis is a set of precisely m sequences of integers. The integers in these sequences are all in the range 1 to n, and a particular integer appears at most once in some sequence.

As one might expect, the sequences will be represented by singly linked lists. In fact we shall use an integer vector, *lt* (for link-table), to store all of the lists which represent sequences in a condis. The fact that an index i into lt is in the kth position of such a list will represent the fact that i appears in the kth position of the corresponding abstract sequence. A separate vector, *sq*, of length m, is used for the heads of the lists. In all cases, zero, which is not a legal condis sequence element, is used to indicate the end of a list; thus, in particular, if sq[j]=0, the jth condis sequence is empty.[5] A separate list of those integers which are not currently members of any sequence is also maintained, and the head of this list is maintained in the simple variable *free*. The following diagram illustrates one possible configuration of a condis object which has been declared with m=3 and n=10:



The full condis form is given below.

```
form condis(n,m:integer) =
beginform
specifications
   requires $n \ge 1 \wedge m \ge 1$;
   let condis = L:{$sq_i$:<$e_{i1}, e_{i2}, \ldots, e_{in_i}$> | $0 \le i \le m-1 \wedge e_{ik}$ is integer};
   invariant $1 \le e_{ik} \le n \wedge \forall i,j \in [0..m-1](e_{ik_1} = e_{jk_2} \supset i=j \wedge k_1 = k_2)$;
   initially $\forall i \in [0..m-1]\ sq_i$ = <>;
   functions
       xtnd(s:condis,i:integer) returns j:integer
           pre $i \in [0..m-1] \wedge \text{SIGMA}_{j \in [0..m-1]}$ length(s.$sq_j$)<n,
           post s.$sq_i$ = <j>~s.$sq_i$',        ! note j is a new value not in any sq (by $I_a$)
       del(s:condis, i,j:integer)
           pre s.$sq_i$ = < . . . , j, . . . > $\wedge$ $i \in [0..m-1]$
           post s.$sq_i$ = <j, . . . >,
       delall(s:condis,i:integer)
           pre $i \in [0..m-1]$
           post s.$sq_i$ = <>,
       full(s:condis) returns t:boolean
           post t = $\text{SIGMA}_{j \in [0..m-1]}$ length(s.$sq_j$) = n;
```

5 We can now explain why the function delall is *not* redundant. The knowledge that zero ends a list is private to condis, and therefore it is not known in symtab. Hence, in the body of leaveblock of symtab, the operation "delall(as,i)" cannot be replaced by "del(as,i,0)". To do so would violate the pre condition of del because if j is a member of sq_i, it means $j \ge 1$.

```
      generator indis(s:condis,i:integer) extends x:integer
         requires 0≤i≤m-1
         let indis = s.sq_i where indis≠<> ⊃
                 (indis = c~<x>~d and c, <x>, and d are disjoint);
         rule for(I, x, <s,i>, ST) =
            premise s.sq_i=c~<x>~d ∧ I(c) {ST} I(c~<x>);
         rule first(P, x, <s,i>, β, S_1, S_2, Q) =
            premise s.sq_i=c~<x>~d ∧ P ∧ ∀y ∈ c(¬β(y)) ∧ β(x) {S_1} Q,
            premise P ∧ ∀y ∈ s.sq_i¬β(y) {S_2} Q;
      auxiliary predicates
         follows(s:condis,i,j:integer) ≡_df ∃k st sq_k = < . . ., i, . . ., j, . . . >,
         mbr(s:condis,i,j:integer) ≡_df sq_i = < . . ., j, . . . >;

   representation
      unique
         sq: vector(integer,0,m-1),
         lt: vector(integer,1,n),
         free: integer
         init begin free ← 1; for i:upto(1,n-1) do lt[i] ← i+1; lt[n] ← 0;
            for i:upto(0,m-1) do sq[i] ← 0 end;
      rep(sq,lt,free) = {SQ_i | 0≤i≤m-1} where
         if sq[i] = 0 then SQ_i = <> else
         if sq[i] = p_1 ∧ (∀j ∈ [1..k-1] lt[p_j]=p_{j+1}) ∧ lt[p_k]=0 then SQ_i = <p_1, . . . ,p_k>;
      invariant
         0 ≤ free ≤ n
         ∧ ∀j ∈ [0..m-1] 0 ≤ sq[j] ≤ n
         ∧ ∀k ∈ [1..n] 0 ≤ lt[k] ≤ n
         ∧ {free, sq[j], lt[k]} = {m+1 0's, 1, 2, ..., n}        ! this term is a multiset equality
         ∧ ∀i ∈ [1..n](succ(free,i) xor ∃!j(succ(sq[j],i)))
            where succ(i,j) ≡_df i=j ∨ (i≠0 cand succ(lt[i],j));

   implementation
      body xtnd in s.free≠0 ∧ i ∈ [0..m-1]
            out (succ(s.free',j) ∧ succ(s.sq[i],j) ∧ s.sq[i]=j ∧ s.lt[j] = s.sq'[i]) =
         begin
         j ← s.free; s.free ← s.lt[j];
         s.lt[j] ← s.sq[i]; s.sq[i] ← j;
         end;
```

```
body del in succ(s.sq[i],j) ∧ i ∈ [0..m-1] ∧ j ∈ [0..n] out (s.sq[i]=j) =
    if s.sq[i]≠j then
        begin local k:integer;
        k ← s.sq[i];
        while s.lt[k] ≠ j do k ← s.lt[k];
        s.lt[k] ← s.free; s.free ← s.sq[i]; s.sq[i] ← j;
        end;

body delall in i ∈ [0..m-1] out (s.sq[i]=0) =
    s.del(s,i,0);        ! a call to the concrete body del, not the abstract function del

body full out (t = (s.free=0)) =
    t ← s.free=0;

formbody indis =
beginform
representation
    rep(s.sq,s.lt,i,x) =
        if s.sq[i] = 0 then <> else
        if x = 0 then c~d where c = s.sq_i and d = <> else c~<x>~d
            where c = <p_1, . . ., p_{r-1}>, x=p_r, d = <p_{r+1}, . . ., p_k>,
            p_1 = s.sq[i], s.lt[p_k] = 0, and (∀j ∈ [1..k-1] s.lt[p_j] = p_{j+1});
    invariant true;
implementation
    body &init out (x=s.sq[i] ∧ (&b ≡ s.sq[i]≠0)) =
        (x ← s.sq[i]; &b ← x≠0);

    body &next in succ(s.sq[i],x) ∧ x≠0 out (x=s.lt[x'] ∧ (&b ≡ s.lt[x']≠0)) =
        (x ← s.lt[x]; &b ← x≠0);
endform

endform
```

The implementation of the four operations in condis should be fairly obvious. *xtnd* merely removes an entry from the free list and places it at the head of the appropriate list; note that this entry is returned (in j) as the value of function *xtnd*. *del* is a bit more interesting. It searches the appropriate list for the entry in lt which points to the first entry, j, which is *not* to be removed. It then moves the entire initial portion of the list to the free space list by simply setting the proper pointers. If all the entries are to be removed, *delall* does this; it calls del to search for the list-ending zero and to move the entire list to the free space list. *full* just tests if the free space list is empty.

The predicate *succ* defined in the concrete invariant is closely related to the abstract predicate *follows*. Although the parameterizations of the two predicates are different, they ask the "same" question and are related by

$$\text{follows}(\text{rep}(sq,lt,free), i, j) = \text{succ}(i, j)$$

The form indis(s,i) defines a generator for elements of the integer sequence s_i, starting with first(s_i). Abstractly, an indis is composed of three (sub)sequences, the first containing the elements already generated, the second the (singleton) current element, and the third the other elements yet to be seen.

In [Shaw76b] we discussed the proof rules for iteration statements. We showed that certain simplifying assumptions about the generator can yield simple proof rules; these assumptions are satisfied by indis, as we will show in the verification of condis. We therefore have a proof rule for the for statement which corresponds closely to Hoare's sequence rule and also a proof rule for the first statement. These proof rules are given in the specifications of indis, and indeed constitute the major part of those specifications. The basis for this specification technique for generators is given in [Shaw76b].

Verification of Condis

We can now verify the form condis.

For the form

1. Representation validity
   Show: $I_c(sq,lt,free) \supset I_a(rep(sq,lt,free))$
   Proof: $1 \le e_{ik} \le n$ holds by the bounds on sq[j] and lt[k] and the fact that the rep function drops the zeroes that indicate the end of a list. The e_{ik}'s are distinct because the multiset {sq[j], lt[k]} contains each of 1, 2, ..., n at most once. The multiset property of I_c implies succ(free,0) and succ(sq[j],0).

2. Initialization
   Show: $n \ge 1 \wedge m \ge 1$ { init } $\forall i \in [0..m-1]\ sq_i = \langle\rangle \wedge I_c$
   Proof: After init we have free=1, lt[1]=2, ..., lt[n-1]=n, lt[n]=0, sq[0]=0, ..., sq[m-1]=0. Using the rep function, each $sq_i = \langle\rangle$ since each sq[i]=0. $n \ge 1$ means $0 \le free \le n$. The bounds on sq[j] and lt[k] and the multiset property are clear. $\forall i \in [1..n](succ(free,i) \wedge \neg succ(0,i))$.

For the function xtnd

3. Concrete operation
   Show: s.free$\neq$0 $\wedge$ i $\in$ [0..m−1] $\wedge$ I_c { body } β_{out} $\wedge$ I_c
   Proof: The four terms of β_{out} are clear as are the bounds in I_c. The multiset property holds because the body permutes the values s.free', s.sq'[i], and s.lt'[s.free']. Since the head of s.free moves to the head of s.sq[i], each i $\in$ [1..n] still satisfies exactly one succ term. β_{in} (and I_c) ensures that the accesses to s.lt and s.sq are within bounds.

4a. β_{in} holds
   Show: I_c $\wedge$ β_{pre} $\supset$ β_{in}
   Proof: i $\in$ [0..m−1] is immediate. If s.free=0, then the multiset property of I_c means, using the rep function, that the SIGMA term is exactly n, a contradiction. Hence s.free$\neq$0.

4b. β_{post} holds
   Show: I_c $\wedge$ β_{out} $\wedge$ β_{pre} $\supset$ sq_i=<j>~sq_i'
   Proof: Since s.sq[i]=j and s.lt[j]=s.sq'[i], the rep function gives sq_i=<j>~sq_i'.

For the function del

3. Concrete operation
   Show: β_{in} $\wedge$ I_c { body } s.sq[i]=j $\wedge$ I_c
   Proof: If s.sq[i]=j then β_{out} holds and I_c is unchanged. If s.sq[i]$\neq$j then define the set G_p = { x | succ(s.sq[i],x) $\wedge$ succ(x,p)}. Add the ghost operation "H←H ∪ {k}" after "k←s.lt[k]" in the while loop and add "H←{k}" after "k←s.sq[i]". A while-loop invariant (placed before the test) is then H=G_k because $G_{s.sq[i]}$ = {s.sq[i]} and

   $$H=G_k \wedge s.lt[k]\neq j \supset H \cup \{s.lt[k]\} = G_{s.lt[k]}$$

   The while terminates because succ(s.sq[i],j) and s.sq[i]$\neq$j. At termination s.lt[k]=j and H=G_k. The multiset property of I_c holds because the last three statements in the body permute the values s.free', s.sq'[i], and s.lt'[k]. Furthermore, each element in H is now a successor of s.free rather than of s.sq[i]. All other successors of s.sq[i] and all previous successors of s.free remain so, respectively. β_{out} and the bounds in I_c are clear.

4a. β_{in} holds
   Show: I_c $\wedge$ β_{pre} $\supset$ β_{in}
   Proof: Immediate from β_{pre} and I_a for condis.

4b. β_{post} holds
   Show: I_c $\wedge$ β_{pre} $\wedge$ β_{out} $\supset$ sq_i=< j, . . . >
   Proof: Only sq_i changes. sq_i now begins with j and there are no other changes to sq_i.

For the function delall

3. Concrete operation
   Show: $\beta_{in} \wedge I_c$ { s.del(s,i,0) } s.sq[i]=0 $\wedge I_c$
   Proof: β_{in} and the multiset property of I_c imply in holds for s.del. (I_c holds for s.del as required.) The out for s.del gives s.sq[i]=0. I_c after s.del gives I_c after delall.

4a. β_{in} holds
   Show: $i \in [0..m-1] \supset i \in [0..m-1]$
   Proof: Immediate

4b. β_{post} holds
   Show: $I_c \wedge i \in [0..m-1] \wedge$ s.sq[i]=0 $\supset sq_i$=<>
   Proof: Only sq_i changes. s.sq[i]=0 means sq_i=<>.

For the function full

3. Concrete operation
   Show: I_c { t←s.free=0 } t = (s.free=0) $\wedge I_c$
   Proof: Immediate

4a. β_{in} holds
   β_{in} is true

4b. β_{post} holds
   Show: $I_c \wedge \beta_{out} \supset \beta_{post}$
   Proof: t = (s.free=0) = (SIGMA . . . = n) using the multiset property of I_c.

To verify the indis generator, we must first reconstruct the pre and post conditions from the specified proof rules:

&init
   post (&b ≡ s.sq_i≠<>) $\wedge$ (&b $\supset$ x = first(s.sq_i) $\wedge$ c = <>)

&next
   pre mbr(s,i,x)
   post (&b ≡ d'≠<>) $\wedge$ (&b $\supset$ x = first(d') $\wedge$ c = c'~<x'>)

Next, we must show that indis satisfies the *standard aggregate assumptions*:

(a) The indis abstraction is explicated in terms of sequences. The normal empty sequence (<>), concatenation operator (~), and leading element selector (first) are available.

(b) The complete sequence to be generated is s.sq_i, which can be decomposed as indicated in the let clause of indis.

(c) The specifications of &init and &next have the required form.

Furthermore, indis satisfies the *basic generator assumptions* because (a) &init and &next terminate and (b) &init and &next alter only the indis variable x (and the return value &b).

Since "sq", "lt", and "free" are unchanged by indis, the I_c of condis still holds and will be used in the proof.

For the form (indis)

1. Representation validity
   Show: $I_c \supset I_a$, i.e., true $\supset$ true
   Proof: Immediate

2. Initialization
   Show: $0 \le i \le m-1$ { } true $\wedge$ true
   Proof: Immediate

For the function &init

3. Concrete operation
   Show: true { x←s.sq[i]; &b←x≠0 } x=s.sq[i] $\wedge$ (&b ≡ s.sq[i]≠0)
   Proof: Clear

4a. β_{in} holds
   β_{in} is true

4b. β_{post} holds
   Show: x=s.sq[i] $\wedge$ (&b ≡ s.sq[i]≠0) $\supset$
   (&b ≡ $s.sq_i$≠<>) $\wedge$ (&b $\supset$ x = first($s.sq_i$) $\wedge$ c = <>)
   Proof: From the rep function for indis, $s.sq_i$ = (if s.sq[i]=0 then <> else some non-empty sequence). Hence &b ≡ s.sq[i]≠0 ≡ $s.sq_i$≠<>. For the second term of the conclusion, assume &b. Then x=s.sq[i]≠0 and the final clause of rep gives $s.sq_i$ = c~<x>~d. Since x=s.sq[i]=p_1, then c = <> whence also x = first($s.sq_i$).

For the function &next

3. Concrete operation
   Similar to &init.3

4a. β_{in} holds
   Show: mbr(s,i,x) $\supset$ succ(s.sq[i],x) $\wedge$ x≠0
   Proof: mbr(s,i,x) means x≠0 by I_a for condis. The term succ(s.sq[i],x) follows from mbr(s,i,x), the rep function, and the definition of succ.

4b. β_{post} holds
   Show: mbr(s,i,x') $\wedge$ x=s.lt[x'] $\wedge$ (&b ≡ s.lt[x']≠0) $\supset$
   (&b ≡ d'≠<>) $\wedge$ (&b $\supset$ x = first(d') $\wedge$ c = c'~<x'>)

Proof: $mbr(s,i,x')$ means $x' \neq 0$ and $s.sq_i \neq <>$, and therefore by the rep function also $s.sq[i] \neq 0$. Hence in the final clause of the rep function, $\&b \equiv s.lt[x'] \neq 0 \equiv d' \neq <>$. For the second term of the conclusion, assume $\&b$. Then $x = s.lt[x'] \neq 0$ and the final clause of rep gives $s.sq_i = c \sim <x> \sim d$ and, because $x' \neq 0$, also $s.sq_i = c' \sim <x'> \sim <d'>$. Since $x = s.lt[x']$, it follows that $x = first\ (d')$ and $c = c' \sim <x'>$.

QED

Examples of the Use of Symtab

In this section we shall present a skeletal example which involves three different styles of usage of the symtab abstraction. It is not our intent either to make this example complete or to suggest that the utility of the abstraction is limited to these three cases. Rather, we wish to bolster the reader's intuition about ways in which the abstraction might be used.

The example we have chosen is a multi-pass compiler for an Algol-like (i.e., block-structured) language, and indeed we have restricted ourselves to the first two passes -- lexical and syntactic analysis, respectively. In this scheme, the first pass is responsible for reading units of the source file (identifiers, literals, punctuation marks, etc.) and converting them to an internal form called a "lexeme". These lexemes are written onto a file which will be read again by the second pass. The second pass is responsible for reading the file of lexemes generated by the first pass and performing syntactic analysis. Although it is not important to our example, the output of the second pass will likely be some other intermediate representation (e.g., reverse polish or trees) which is suitable for optimization and code generation.

Here, then, is the skeletal program; more detailed comments on the uses of the symtab abstraction, and on the program in general, follow the example.

```
function compiler (source: file(char))=
   begin
       form condis . . .;
       form symtab . . .;
       form id extends string=
           beginform
           specifications
               function hash (s:id, m:integer) returns k:integer pre m>0 post 0≤k<m';
           . . .
           endform;
```

```
        form lex extends integer=
            beginform
            specifications
                function hash (x:lex, m:integer) returns k:integer pre m>0 post 0≤k<m';
            . . .
            endform;

        local L: file(lex);

        begin   ! pass 1
            local NT: symtab (id, 127, 1000);
            !
            ! pure lexical pass, see discussion below.
            !
        end;

        begin   ! pass 2
            form attributes = . . .       ! see discussion below
            local A: vector (attributes, 1, 2000);
            local ST: symtab (lex, 127, 2000);
            !
            ! syntactic (parse) analysis pass; see discussion below.
            !
        end;

        . . .
    end;
```

This program first defines four forms. Symtab and condis have been defined in detail previously and hence are not repeated. The forms *id* and *lex* are extensions of strings and integers, respectively, and merely add hashing functions; we have not defined the implementations of these functions, since they are not germane to the example. Note too that a file of *lex*es is defined at the outermost block level; this file is the explicit interface between the first and second passes.

As noted earlier, the function of the first pass is to convert the external representation of the program (a file of characters) into a more convenient internal form -- namely a file of lexemes (where each lexeme represents an atom of the language). Since this pass does no syntactic analysis, in particular it does not recognize block structure. This implies that all occurrences of the same atom (e.g., "xyz") will be mapped to the same lexeme. This mapping is accomplished through the use of the NT (for name-table) instantiation of symtab; indeed, the *only* use of NT is to obtain this unique mapping and the instantiation is therefore deleted on exit from the block in which the first pass is accomplished.

In skeletal form, the body of the block for pass 1 might look somewhat as follows:

```
open(source); open(L);
while ¬end of file(source) do
begin
    local i:id, x:lex;
    !
    ! do whatever is appropriate to assemble the next atom
    ! from the source file into "i".
    !
    if (x←lookup(NT,i))=0 ∧ ¬full(NT) then x←insert(NT,i);
    write (L,x);
end;
rewind(L);
```

Note that the operations *enterblock* and *leaveblock* are not used, all *insert* operations are done at the same block level, and only one entry per atom will be made.

The second pass is substantially more complex since it performs the full syntactic analysis; hence we will not even attempt to illustrate its skeletal form. We would, however, like to point out several things about it.

First, notice that this block defines a form named *attributes*. We have not shown the body of this form, since it will be highly language- and machine-specific. However, the notion is that this form provides for the storage and manipulation of whatever information must be retained about a symbol, e.g., its type, run-time storage address, array bounds, and so forth.

Second, we have declared a vector, A, of these attribute objects. As suggested in an earlier section, instances declared at a given block level will be associated with a unique integer, but this integer will be different from the one associated with the same identifier declared at a different block level. These integers will, in turn, be used as indices into the vector A (e.g., to set and retrieve information about the identifier).

Finally, we have declared another instantiation of symtab, ST. This one *will* be used to recognize block structure, and, specifically, will map from the simple lexemes generated in the first pass into indices into the vector, A, of attributes. As the parser detects blocks (begin-end pairs) in the source program, it will invoke *enterblock* and *leaveblock*. The declaration processing routines will invoke *defined* to determine whether an identifier has been declared twice at the same block level (presumably an error), and perform *insert* operations to define the instances of the identifier at the current block level. The rest of the compiler will perform *lookup* operations to obtain the index of the attribute vector entry associated with specific lexemes. (Note, by the way, that by appropriate ordering of *insert* and *lookup* operations the declaration processor can obtain either of the interpretations of "block-structure" discussed in the introduction.)

Before leaving this example, let us return to the form attributes (defined in pass 2) to illustrate another potential use of the symtab abstraction. As was mentioned in the introduction, in general the mapping from identifier to unique integer may be context-sensitive. Block structure is the most familiar form of such sensitivity, but another is name qualification, as in field selectors for records. In many languages one makes a declaration such as

```
x:record(name:string, age:integer, z:integer);
```

and then refers to "x.name", "x.age", and "x.z". A problem arises when, at the same block level, there is another declaration such as

```
y:record(ss:integer, z:boolean);
```

In such a case the identifier "z" is no longer unique -- its interpretation depends upon the name it qualifies.

There are many ways one might treat this, including inserting each of "x", "x.name", "x.age", "x.z", "y", "y.ss", and "y.z" as complete identifiers in ST. An attractive alternative, however, is to include instantiations of symtab in each of the attributes; that is, to make form attributes appear somewhat as follows:

```
form attributes=
   beginform
   . . .
       representation
           . . .
           unique qual:symtab(lex,1,10),
   . . .
   endform;
```

If this is done, then to determine the interpretation of "x.z" one would first search ST for the index, i, associated with the lexeme for "x", then search A[i].qual for the index associated with the lexeme for "z".

Although this compiler example has been sketchy, we hope that it has suggested some of the ways in which the symtab abstraction may be applied. The details of the example are not important, except insofar as they help the reader's intuition; what *is* important is the notion that well-chosen abstractions have many uses. The class of broadly useful abstractions is simply too large to include them all in a single programming language -- hence Alphard has chosen to provide a linguistic facility so that the programmer may define them. Many such (verified) abstractions will find their way into the library, and hence incrementally enhance the "power" available to the programmer -- *without*, at the same time, limiting him to the language designer's preconceived notions of what constitutes an appropriate set of abstractions (or, for that matter, implementations).

Conclusions

A programming language is a tool for the construction and communication of programs; as such its utility should be measured relative to these tasks. In other words, the language should be *used*, and the quality of that use must be judged. While this is true of any programming language, it is especially so of one such as Alphard, which departs substantially from those in common use.

Thus, in this and other reports we are attempting to exhibit Alphard in relatively realistic contexts and, along with the reader, to judge the practical utility of our creation. It is far too soon to draw definitive conclusions -- that must await the use of Alphard in real programs -- but we would like to share some of our impressions resulting from these experiences.

First, the symtab abstraction is about the (conceptual) size we envision for most abstractions; larger programs will be constructed by further "layering". Thus we take our ability to specify and verify this form as fairly strong evidence that larger programs will also be tractable.

Second, in most respects the implementation is a practical, efficient one. This reinforces our intuitions that *no* efficiency need be sacrificed to obtain clear, verifiable programs. (The one exception to this statement is our use of fixed-sized vectors and, correspondingly, integers for the unique identification of symbols. A more realistic implementation would, perhaps, have done true dynamic storage allocation and used references. We avoided this implementation primarily because it would have carried us into portions of Alphard not covered in previous reports, but also because those portions of the language are still in flux. We trust that the reader will forgive this departure from realism.)

Third, one of the anticipated advantages of an Alphard-like language is that a library of verified abstractions will develop. Both of the forms developed here might well go into that library so we are getting some evidence that this hoped-for advantage will be realized.

Fourth, one of our private objectives was to make the form mechanism strong enough to support an extremely broad class of abstractions -- the ultimate target being the spectrum covered by our intuitive notion of the word "abstraction". The evidence is not conclusive, but we are feeling better about meeting that goal all the time.

Finally, we should say a few words about our experience concerning the effort needed to define a form. It should be clear that the actual code in a form body, i.e., the implementation part, is roughly the same size as the corresponding code in other languages (although the first statement does seem to shorten many of the examples). Moreover, for some reason, the information needed for verification (abstract and concrete invariants, abstract pre and post conditions, rep function, etc.) usually seems about equal to the code size; thus a full form is about twice the size of the code alone. This does not particularly concern us, since these kinds of specifications tend to replace much of the documentation that would otherwise be needed -- and they are certainly more precise.

We find the verification of a form, once the specifications and code have been written, to be more difficult and time-consuming than coding, but not unreasonably so (say by as much as a factor of two or three). Sometimes it is necessary to modify the specifications, or the code, during the verification in order to remove inconsistencies that are uncovered. The verification may also suggest different specifications, usually ones that are more constrained but sometimes simpler ones. In spite of the difficulties, the bodies of functions tend to be small and their proofs correspondingly small, as can be seen from these examples. Moreover, the proofs of the two forms symtab and condis were independent. To date our proofs have been manually generated, but we envision having automated, interactive aids in the future. These should reduce the verification time to approximately the coding time. Since this is less than the time currently spent on debugging, we feel highly encouraged.

The majority of our time goes into designing and specifying the abstraction. There are two related aspects of this: getting the intuitive abstraction "right", and formalizing it (at least sufficiently for it to be verified). The two appear related in that difficulty in formalizing an intuitive abstraction often seems to uncover muddy thinking at the intuitive level. While we seem to be improving our ability to formalize, indicating that it is a learnable skill, we have no easy rules for picking the right abstraction in the first place. While, with practice, our abilities in choosing abstractions may also improve, we suspect that this is a fundamental problem of design and has a significant aesthetic component.

It is clear that we are just learning to use the power of the tools we are creating and exploring. Much remains to be discovered about what is possible or impossible, easy or hard, and reasonable or unreasonable to do with the facilities. In this connection we note that an early version of symtab was a one-level form, used no generator such as indis, and had only some of the same verification information. Although that version of symtab used the same implementation ideas, it was essentially incomprehensible. When we realized that multiple ideas were becoming confused, we separated the maintenance of the lists from the lookup algorithms. The result was that the code, the specifications, and the verification all became much more manageable.

Acknowledgements

We owe a great deal to our colleagues at CMU and ISI, especially Mario Barbacci, Neil

Goldman, Donald Good, John Guttag, Paul Hilfinger, David Jefferson, Anita Jones, David Lamb, David Musser, Karla Perdue, Kamesh Ramakrishna, and David Wile. We would also like to thank James Horning and Barbara Liskov and their groups at the University of Toronto and Massachusetts Institute of Technology, respectively, for their critical reviews of Alphard.

References

[Gries71] David Gries, *Compiler Construction for Digital Computers*, Wiley, 1971.

[Guttag76] John Guttag, "Abstract Data Types and the Development of Data Structures", *Supplement to the Proceedings of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure*, March 1976 (pp. 37-46). Also *Communications of the ACM* (to appear).

[Halmos60] Paul R. Halmos, *Naive Set Theory*, Van Nostrand, 1960.

[Hoare72] C. A. R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica*, 1, 4, 1972 (pp. 271-281).

[Knuth73] Donald E. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison-Wesley, 1973.

[Shaw76a] Mary Shaw, "Abstraction and Verification in Alphard: Design and Verification of a Tree Handler", *Proc. Fifth Texas Conference on Computing Systems*, 1976 (pp. 86-94).

[Shaw76b] Mary Shaw, Wm. A. Wulf, and Ralph L. London, "Abstraction and Verification in Alphard: Iteration and Generators", *Carnegie-Mellon University* and *USC Information Sciences Institute Technical Reports*, 1976. Also *Communications of the ACM* (to appear).

[Wegbreit76] Ben Wegbreit and Jay M. Spitzen, "Proving Properties of Complex Data Structures", *Journal of the ACM*, 23, 2, April 1976 (pp. 389-396).

[Wulf76a] Wm. A. Wulf, Ralph L. London, and Mary Shaw, "Abstraction and Verification in Alphard: Introduction to Language and Methodology", *Carnegie-Mellon University* and *USC Information Sciences Institute Technical Reports*, 1976.

[Wulf76b] Wm. A. Wulf, Ralph L. London, and Mary Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Transactions on Software Engineering*, SE-2, 4, December 1976 (pp. 253-265).

Appendix A
Informal Description of Verification Methodology

Alphard's verification methodology is designed to determine whether a form will actually behave as promised by its abstract specifications. The methodology depends on explicitly separating the description of how an object behaves from the code that manipulates the representation in order to achieve that behavior. It is derived from Hoare's technique for showing correctness of data representations[Hoare72].

The abstract object and its behavior are described in terms of some mathematical entities natural to the problem domain. Graphs are used in [Shaw76a] to describe binary trees; sequences are used in [Wulf76a,b] to describe queues and stacks and in condis to describe list processing, and so on. We appeal to these abstract types

- in the invariant, which explains that an instantiation of the form may be viewed as an object of the abstract type that meets certain restrictions,

- in the initially clause, where a particular abstract object is displayed, and

- in the pre and post conditions for each function, which describe the effect the function has on an abstract object which satisfies the invariant.

The form contains a parallel set of descriptions of the concrete object and how it behaves. In many cases this makes the effect of a function much easier to specify and verify than would the abstract description alone.

Now, although it is useful to distinguish between the behavior we want and the data structures we operate on, we also need to show a relationship that holds between the two. This is achieved with the representation function rep(x), which gives a mapping from the concrete representation to the abstract description. The purpose of a form verification is to ensure that the two invariants and the rep(x) relation between them are preserved.

In order to verify a form we must therefore prove four things. Two relate to the representation itself and two must be shown for each function. Informally, the four required steps are[6]:

[6] We will use I_a(rep(x)) to denote the abstract invariant of an object whose concrete representation is x, I_c(x) to denote the corresponding concrete invariant, italics to refer to code segments, and the names of specification clauses and assertions to refer to those formulas. In step 4b, "pre(rep(x'))" refers to the value of x *before* execution of the function. A complete development of the form verification methodology appears in [Wulf76a,b].

For the form

1. Representation validity
   $I_c(x) \supset I_a(\text{rep}(x))$

2. Initialization
   requires { *init clause* } initially(rep(x)) $\wedge I_c(x)$

For each function

3. Concrete operation
   $\underline{\text{in}}(x) \wedge I_c(x)$ { *function body* } $\underline{\text{out}}(x) \wedge I_c(x)$

4. Relation between abstract and concrete
   4a. $I_c(x) \wedge \underline{\text{pre}}(\text{rep}(x)) \supset \underline{\text{in}}(x)$
   4b. $I_c(x) \wedge \underline{\text{pre}}(\text{rep}(x')) \wedge \underline{\text{out}}(x) \supset \underline{\text{post}}(\text{rep}(x))$

Step 1 shows that any legal state of the concrete representation has a corresponding abstract object (the converse is deducible from the other steps). Step 2 shows that the initial state created by the representation section is legal. Step 3 is the standard verification formula for the concrete operation as a simple program; note that it enforces the preservation of I_c. Step 4 guarantees (a) that the concrete operation is applicable whenever the abstract pre condition holds and (b) that if the operation is performed, the result corresponds properly to the abstract specifications.