List Processing and Object-Oriented Programming Using Forth

Dennis L. Feucht

Innovatia Laboratories

Abstract

The list data structure has significant advantages for symbolic computing and can be implemented easily in Forth. Primitive operations in Forth, list-space management, and list structures for object-oriented programming are briefly presented, as well as an overview of semantic networks, some of their logic, and advantages of implementing them in Forth.

Some unique advantages of Forth for object-oriented programming are suggested.

Introduction

In numeric computing, data structures often are implemented by allocating a fixed block of memory. Variables and arrays are fixed in length at compile time and are accessed efficiently with the linear addressing scheme found in digital computers. For symbolic computing, data access by linear addressing is inefficient due to data complexity and random structure. Furthermore, symbolic data continually grows and shrinks during program execution, requiring run-time management of memory. This dynamic management is usually not needed for data structures based on linear access, such as arrays or stacks, because a fixed block of memory can be allocated for a given structure at compile time based on knowledge of its maximum memory requirement during execution.

The List Data Structure

An especially versatile data structure that at least partially alleviates the constraint of linearly addressed memory is the list, an ordered collection of items. Lists can be used to implement stacks, arrays, trees, and other arbitrary structures while simplifying dynamic memory management. The use of the LISP (LISt Processing) language for symbolic programming exemplifies the value of basing data structures on lists — and specifically, singly-linked lists. This structure is familiar in Forth implementations. The dictionary of Forth words is a singly-linked list where the link-field of the word defined last points to the name-field of the next-to-last word, etc. It is typical also to find vocabularies and initialized words linked together in Forth.

The use of lists in (non-tokenized) Forth are, however, not adequate for symbolic computing because the words or vocabularies are linked at compile-time and stay where they are put. It would not be feasible, say, to expand the size of a word in the middle of the dictionary by moving the latter part of the dictionary up in memory to make room. All the pointers to the moved words would then require updating, and this generally would be a massive task.

A Forth Implementation of Lists

A solution to this memory management problem is to allot space in the dictionary where lists can be built and managed separately. Several such list-spaces could be created. Lists are then built out of primitive structural elements called **CONS**-cells (after LISP). These cells are illustrated in Figure 1, where five of them are used to build a list. They consist of three fields, shown in Figure 2. At the pointer, a, to the cell is a field for reference counts (explained later). The next field, the head, contains a pointer to the first item of the list; the last field, the tail, contains a pointer to the next cell in the list, and acts as a link-field.

Lists are graphically represented in at least two ways, shown in Figure 1. Fig. 1a is a block diagram of list structure, and is the most explicit. Fig. 1b shows the same list in parenthesis notation, which is textually more convenient. To mark the end of a list, the tail of the last **CONS**-cell is pointed to **NIL**. (This is the convention in LISP.) Besides marking the end of lists, **NIL** also represents the empty list, which, in parenthesis notation, is (). In Forth, **NIL** can be implemented as a variable.

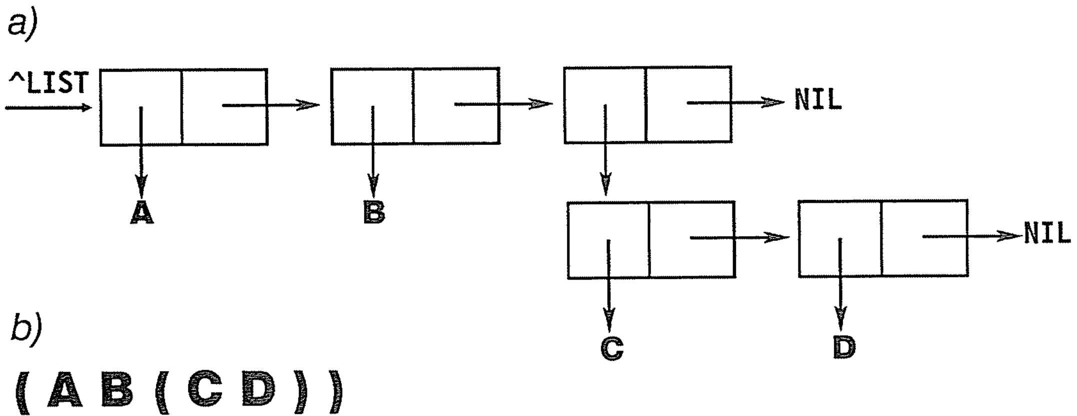

Fig. 1 Graphical representations of list structure: (a) block notation, showing explicitly memory locations and pointers. **NIL** indicates end of list. (b) parenthesis notation, a text-based denotation of lists where parentheses enclose lists. The list shown here has three items, the third being a list containing two items.

Forth variables are also a straightforward way of creating the names of list items pointed to from **CONS**-cells in list-space. They can also be used to name lists, where the value of the variable is a pointer to the first cell (or head) of the list identified by the variable name. Such a variable is called a list-identifier (or list-id). (LISP has a hidden "dictionary" containing item and list names called the object list or oblist.) An implementation of the list (A B) in Forth using this scheme is shown in Figure 3. Since **NIL** is a list yet resides outside of list-space, it is the only exception requiring special handling in list-manipulation. Consequently, **NIL** must point to itself so that (among other reasons) it will terminate in **NIL**, as all lists must.

List Data Types

While the data types of Forth will be used to implement a LISP-like extension to it, lists themselves need a few data types. So far, lists are one data type. The items of a list can be other lists or names of items. These names have been implemented as Forth variables, but to distinguish them from lists, they are called atoms (as in LISP). **NIL** is the only atom which is also a list. To distinguish between a list and an atom, pointers outside of list-space point to atoms. That is, any word in the Forth dictionary is, relative to lists, an atom. Symbolic expressions or S-expressions

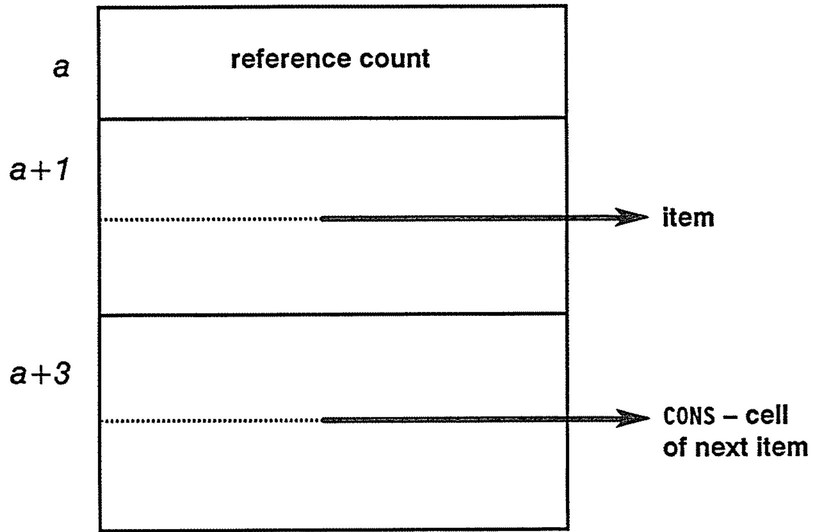

Fig. 2 Implementation of a CONS-cell, the elemental structure of lists. The field at address *a* contains the number of pointers to the cell; at *a*+1 is the pointer to the item, and at *a*+3, the link pointer to the next cell in the list. Each block is a byte for a 5-byte cell size for 8 bit microcomputers.

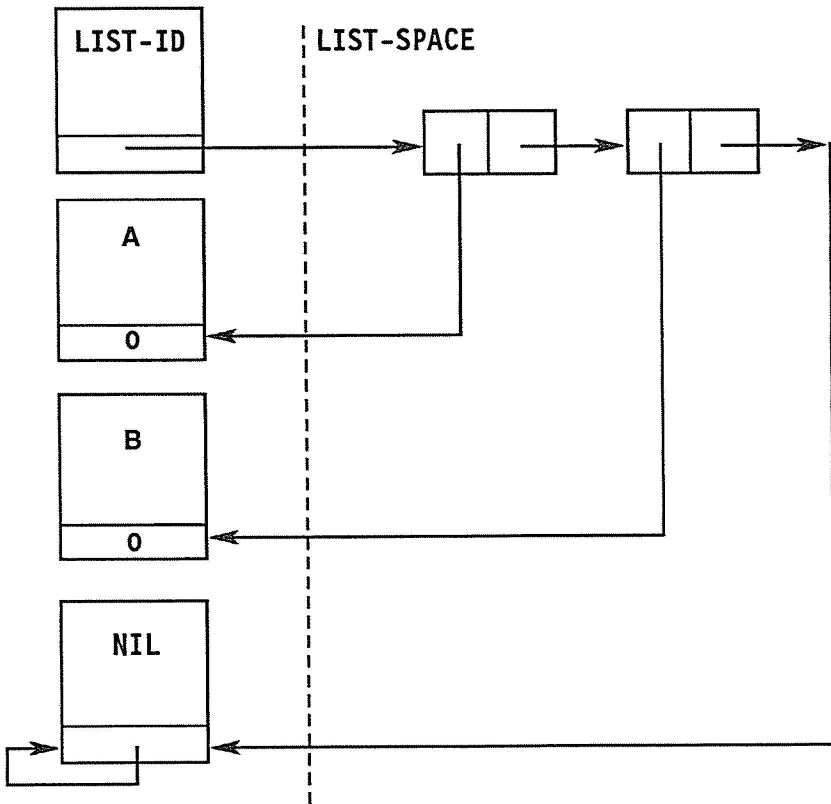

Fig. 3 A list identifier, LIST-ID, defined as a Forth variable, is an atom. Its value is a pointer to the head of the list (A B). A, B, and NIL are also atoms, and are defined as Forth variables. NIL is a special case since it is the only list (the empty list) outside of list-space. NIL points to itself, so that, as a list, it terminates in NIL.

are lists or atoms. Given an address, A, the word **ATOMSP** returns a false flag if A lies within list-space. The word **ATOM**, acts similarly. Another word, **@L**, is needed because lists can be given as arguments in two forms: the list pointer itself or the list-id. The word **@L** takes an address, A, and determines if it is an atom. If so, it points to a list-id, so **@L** takes the value of the the list-id (variable) as the pointer to the list itself. If A already lies within list-space, it is already a list pointer and is unaltered. With these words, the list-oriented data types can be easily handled in the Forth environment.

Primitive List Operations

The list-handling functions found in LISP can be implemented easily in Forth. LISP uses prefix notation while Forth uses post-fix (or "reverse Polish"). This syntactic difference is not significant; by maintaining the order of the arguments as in LISP, equivalent Forth words can be used with negligible confusion to perform the same functions as those similarly named in LISP. The arguments for these words are, in Forth, passed on the stack. (LISP has a hidden stack inaccessible to the programmer.) Like Forth, LISP uses a small set of primitive functions out of which other "high-level" functions are built. A few of these will be presented here to illustrate their implementation in Forth.

To get into lists, the words **FIRST** (or **CAR**) and **TAIL** (or **CDR**) are used. **FIRST** takes a pointer to a list and returns the first item of the list:

```
( L - S) : FIRST  @L DUP NIL = NOT IF 1+ @ THEN ;
```

FIRST takes L, converts it to a list pointer with **@L**, and checks for **NIL** as an exception list. If L is not **NIL**, L is incremented to point to the head of the **CONS**-cell pointed to by L. (In this implementation for 8-bit computers, the reference count field is one byte; hence the **1+** indexing to the head.) At the head, the pointer there to the first item of the list is fetched. To return the pointer to the rest of the list beginning with the second item (or the tail of the list), TAIL is used:

```
( L1 - L2) : TAIL  @L DUP NIL = NOT IF 3 + @ THEN ;
```

It is similar to **FIRST** except the pointer in the tail of the first CONS-cell is returned.

To construct lists, the word **CONS** is used and it requires consideration of memory management. To build list structure, a means of obtaining unused (or free) **CONS**-cells is needed. The word **GETCELL** does this. It returns a pointer to an available cell. To find a free cell, **GETCELL** searches list-space in some manner looking for a cell with a reference count of zero. The reference count indicates that there are no pointers to that cell, and thus it must be free. **CONS** is:

```
( S L1 - L2) : CONS  GETCELL DUP >R SWAP OVER 3 + ! 1+ ! R> ;
```

CONS gets a free cell and places S, the item, in the head of the cell, and links the cell to L1 by placing L1 in the tail. It then returns a pointer to the new cell, which is now at the head of the list. For the first cell in a proper list, L1 must be **NIL**.

The final primitive list-handling word to be given here sets a list-id to point to a list. **SETQ** takes a list-id, I, and list pointer, L, and returns L after setting I to point to L:

```
( I L - L) : SETQ  @L SWAP DUP @L >R OVER SWAP ! DUP RC+ R> RC- ;
```

SETQ illustrates how reference counts are managed. The word **RC+** takes a list pointer and goes down the list, incrementing the reference count of each cell in the list. **RC-** decrements instead. Because I might have pointed to a previous list, this list is decremented by **RC-**. Then list L is incremented by **RC+** since I now points to it (as well as other possible pointers).

List-Space Management

By using reference counts, any list operations (such as SETQ) that modify pointers to lists must update reference counts. In this way used memory is kept track of as list words execute, providing an incremental management of list-space. List memory management is called garbage collection. An alternative garbage collection scheme is the mark and sweep technique. All list-id pointers are used to trace out used cells and they are marked. Any remaining cells are then linked into a free-list from which free cells can be obtained by GETCELL. This garbage collector runs whenever the free list becomes empty, and can take a long time to complete. Because real-time programs such as expert operators or robot planners cannot afford a pause for several seconds, the incremental garbage collection of reference counting is an attractive alternative. The drawbacks to reference counting are that it is less efficient overall and causes the program to run slower due to management overhead. Also, improper lists (those not ending in NIL) such as circular lists, and destructive list operations, pose difficulties. The mark and sweep technique is attractive with a large memory, since some programs may never invoke garbage collection and yet run faster.

Data Structures Built out of Lists

LISP provides some functions for handling simple list-based data structures. The most common is the property list. Property lists are of the form:

```
( -1 P1 V1 P2 V2 ... PN VN )
```

where the -1 is a number not used by the system as an address (and consequently is implementation-dependent) followed by alternating property/value pairs. Atoms can have property lists. Since item names are atoms, and because they are being created as Forth variables, the variable values can point to property lists. This scheme makes it easy to implement associative databases. A property list access function is GET, which takes an atom, I, and property, P, and returns a pointer to the value of the property. If the property is not found, NIL is returned. PUT takes an atom, value, and property, and sets the property value. If the property does not exist, it is added to the list and given the value provided. Finally, REMPROP removes property/value pairs when given the atom and property name.

Another structure-building scheme common to LISP is a set of words that access association lists or A-lists. These lists have the structure:

```
((P1 V1) (P2 V2) ... (PN VN))
```

and are accessed by ASSOC. It takes a property and list, and returns a pointer to the property/value pair, which is a list. Unlike the "flat" property list, A-lists have one level of nesting, where property/value pairs are kept in lists. This structure uses more CONS-cells than property lists but access is faster. A-lists and property-lists are structures that can be used to implement knowledge structures. Knowledge engineering problems begin by identifying the domain of expertise that will be represented. Then an appropriate theory of representation is applied to the domain to produce a knowledge representation that contains knowledge structures in the language of the domain. Finally, this representation is implemented using a suitable computer language and the data structures it supports. Choice of a computer language is based partly on how well the given knowledge structures are supported by the data structures of the language. For example, some knowledge structures are semantic networks, frames, and blackboards. Lists are well-suited for implementing most of these structures effectively.

Generally, those things which are represented from the knowledge domain, called objects, are most effectively implemented by object-oriented programming languages. These languages provide a powerful environment for creating and managing data structures that embody

knowledge of objects and their relationships. Some languages, such as Smalltalk, are explicitly designed to handle objects, while others, like LISP, provide a general environment where structures for objects can be easily created out of lists. Logic programming languages like Prolog, combine some of the list-handling versatility of LISP with a relatively fixed overall structure for implementing knowledge. In Prolog, knowledge is a flat list of clauses, which are facts or rules of first-order logic. In simple Prolog, grouping of knowledge into chunks that can be more efficiently indexed is not possible. To call a language object-oriented implies that knowledge can be organized relatively free from language constraints.

Object-Oriented Programming Using Semantic Networks

Semantic networks are a common form in which knowledge is represented in natural language understanding and robot applications. Figure 4 shows a part of a semantic net used to represent a simple robot world. A class of objects, wall, contains some general properties of walls: border and color. These properties in turn have general values; walls are generally white, for example. A particular kind of wall, outer-wall, has a property specific to its kind, namely, window. Left-wall is a specific wall that is an outer wall. However, it has some properties that are exceptions to the general case. Its color is red rather than the default value of white, and it has no window. In semantic nets, the local properties of an object cancel the default values that are generally true of objects belonging to the same class. This default logic is not the same as the first-order logic used in Prolog, and is less well-defined but often more adaptable for representation of some domains. In first-order logic, no exceptions can exist for universally quantified variables, while in default logic, property cancellation can cause exceptions.

Some of the ill-defined nature of semantic networks can be seen in the ambiguity of the is-a relationship. This is the most common relationship between objects in typical semantic net representations. For case 1 in Fig. 4, is-a denotes that left-wall is a member of the set outer-wall. In Prolog, this would be:

```
outer-wall(left-wall)
```

or, "left-wall is an outer-wall." In the second case of Fig. 4, is-a denotes that the set outer-wall is a subset of wall and is a universally quantified conditional, a rule of the form:

```
wall(X) :- outer-wall(X)
```

or "for all X, if X is an outer-wall, then X is a wall." The two meanings of is-a are distinct, rendering it ambiguous in this example (and in most).

Forth and Object-Oriented Programming

At first, it may appear that Forth is not only not an object-oriented language but that it also is not fit for use in object-oriented programming. While Forth has found limited use in knowledge engineering so far, it has several major features which make it a contender for consideration. First, it is similar to LISP in some important respects. Both are functional languages, passing arguments to functions which return results. Both are extensible in that new functions can be added. They are both naturally recursive and have simple syntax. They are both small languages when non-extended. (LISP is commonly believed to be large because of the substantial extensions often made to it.) Both provide complete computing environments and can run apart from an operating system if preferable. Finally, both are very interactive and appeal to an AI style of programming, where it is easy to experiment quickly in bottom-up fashion with new ideas. Both LISP and Forth programs tend to be short and consequently must be well thought out. Finally, special-purpose processors for both languages are commercially available, providing the process-

ing power to solve practical problems. All these similarities of Forth with LISP are advantageous for object-oriented programming.

Besides these features, Forth has some unique advantages. First, the dictionary assumes the form of a tree of linked lists when multiple vocabularies are used. Vocabularies can represent objects in semantic nets. Control of property inheritance can occur through vocabulary search order. Using the **ONLY** and **ALSO** extensions of Forth-83, the order in which vocabularies are searched for a given property will determine precedence. For example, in Fig. 4, `wall` and `outer-wall` can be defined as vocabularies containing their respective properties as words. The is-a relationship of `outer-wall` to `wall` can be established by the vocabulary-ordering word `outer-wall!`:

```
: outer-wall!  ONLY wall ALSO outer-wall ;
```

`Outer-wall!` makes `wall` the class to which `outer-wall` belongs since it is searched for default properties after `outer-wall`. To establish `left-wall` as belonging to `outer-wall`, define:

```
: left-wall!  outer-wall! ALSO left-wall ;
```

`Left-wall` inherits properties from `outer-wall` which in turn inherits properties from `wall`. Since `left-wall` is searched first, a property found in it will take precedence over `outer-wall` and `wall`.

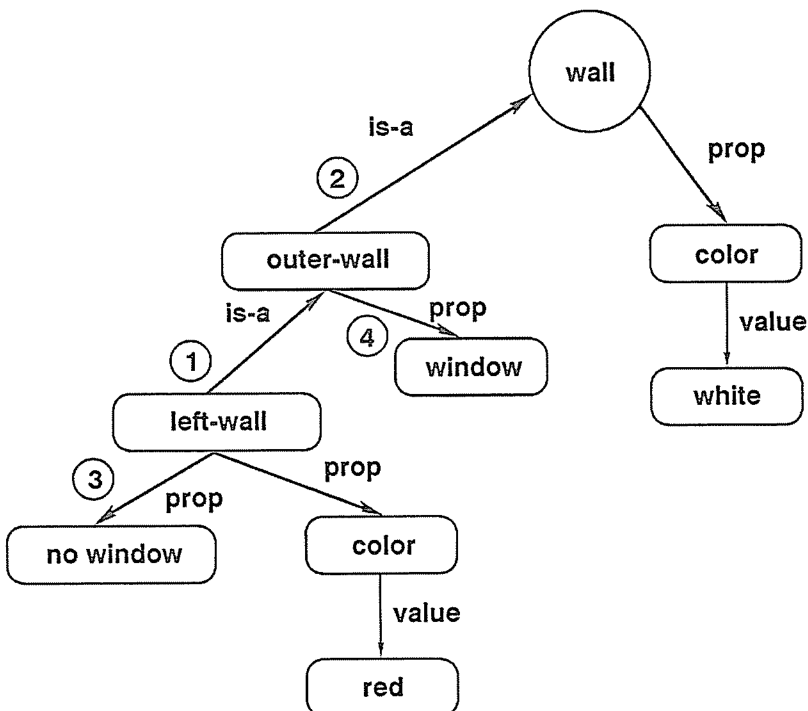

Fig. 4 An example of a semantic network representation for part of the world of a robot. Relationships between nodes are represented by labelled arcs. Besides property and value relationships, is-a (1) expresses membership of left-wall in the class outer-wall and is-a (2) is the conditional: outer-wall(X) :- wall(X), demonstrating the ambiquity of is-a in common use. Properties are inherited through both kinds of is-a relationships by default logic, and exceptions for an instance of a class are possible. Left-wall has the color of red as a locally defined relationship rather than the inherited color of white.

By using vocabularies with separate list-spaces, independent knowledge sources can be established. The same atom names can then be used in different contexts in the different knowledge sources. This scheme could form the basis of a blackboard system, where multiple knowledge sources contribute to a global database (the blackboard) and act on data on the blackboard when triggered by it.

The main disadvantage — and also advantage — of Forth is that it does not contain memory management as part of the language. This is a major departure from LISP. Because the dictionary is statically allocated, Forth itself suffers from the constraints of linearly adressable memory. However, by not including memory management, it runs faster and can accommodate a variety of management schemes. Also, if Forth is implemented as token-threaded code, the entire dictionary can be managed by standard schemes (such as that found in the Macintosh operating system) except for word headers (corresponding to the oblist in LISP). Such a Forth implementation would allow selective forgetting of words within the dictionary. Furthermore, a completely managed Forth is possible by implementing Forth entirely out of `CONS`-cells, including name-field strings. Some of the recent techniques for the efficient implementation of strings in LISP could be borrowed for such a list-based Forth. While the address interpreter would contain an extra indirection or two, it would not be substantially slower than indirect threaded-code implementations (except for garbage collection overhead).

Summary

List processing in Forth is not difficult to implement and provides a computing environment for object-oriented programming and knowledge engineering. Forth compares favorably to other languages used for knowledge engineering, though it has some surmountable drawbacks. Some of its unique advantages are attractive for semantic net and blackboard representations. With the availability of Forth engines, its use in knowledge engineering is inevitable.

References

Designing and Programming Personal Expert Systems, Carl Townsend and Dennis Feucht, TAB Books, Blue Ridge Summit, PA, 1986.