FORTH Implementation in a High-level Language

James C. Bender
The BDM Corporation
10260 Old Columbia Road
Columbia, Maryland 21046

A prototype FORTH interpreter has been implemented in Ada as a research project. The FORTH interpreter is written to take advantage of Ada features, such as range checking, exception handling, and utility packages for string handling and numerical operations. In the past, FORTH interpreters have been written in high-level languages; most have been source-level interpreters. That is not true in this case; however, in this case, a core vocabulary is implemented as of a set of Ada procedures. Each procedure corresponds to a single intermediate language code. The intermediate code instruction set is, therefore, very large. Efficiency is improved with a higher level instruction set, rather than with a primitive instruction set.

The components of the interpreter are contained in a set of Ada packages. The central core of the interpreter consists of a package which defines the dictionary, a set of registers, some basic operations, such as pushing to and popping from the operand and return stacks, and pushing an item to the dictionary. The dictionary and stacks reside in an integer array. A hashed table is used for storing word names and pointers into the dictionary. A table lookup is performed rather than a search through a linked list in the dictionary when a word name is to be checked. This gives considerably improved performance when executing in a purely interpretive mode, rather than when executing compiled words. A problem with this approach is that vocabularies become more difficult to implement.

A special feature of the interpreter allows the entire dictionary and register environment to be saved and reloaded by name. A word is defined which, followed by a file name, saves the dictionary, name table, and registers to the named file. Another word followed by the file name will cause the environment, previously saved, to be reloaded. Reloading the environment, rather than loading a file which must be compiled, saves a considerable amount of time during startup.

The keys to the functioning of the interpreter are the word definition format in the dictionary and the inner interpreter algorithm. Both are closely related issues.

The word definition contains a flag indicating that the word is a primary or secondary and another flag indicating that the word is an immediate word or a normal word. Primary words contain intermediate code terminated by an "end tag." The "end tag" is a special operation code denoting the end of the word. Secondary words contain a list of word addresses, also terminated by the "end tag." The format of a word definition is as follows:

a. Word N: length of name and first character;

b. Words (N + 1) to (N + 6): two packed characters;

c. Word N + 7: link to last word definition;

d. Word N + 8: tag field;

e. Word N + 9: start of code; and

f. Word N + J (where J - 10 is the length of the code): end tag.

The inner interpreter (see Figure) is called with a pointer to the word to be executed. If the word is a primary word, a procedure which executes intermediate codes is called. If the word is a secondary word,the procedure, which executes secondary words, calls the procedure, which in turn executes words, for each word address in the code body. One additional problem exits: some words increment the pointer to the current word so that the pointer is reset past data imbedded in secondary word definitions. This resolved by setting and resetting the instruction pointer in the environment definition package. Word type and execution mode type checking are done with a simple table look-up. Word type (primary or secondary) and execution mode (immediate or normal) are encoded, rather than specific bits being set.

The outer interpreter is written in Ada and includes the lexical analysis, compilation, and invocation of the inner interpreter for execution. The outer interpreter is implemented as a state-transition network with a state variable and case statement. This architecture allows a control path, which is a non-planar graph, to be implemented in a structured way. Instead of a "compile" mode or an "execute" mode, the differentiation is made by the current position in the outer interpreter state-transition graph. If an error occurs, the state variable is reset to the starting position in the graph. The use of an Ada string-handling package simplifies the text-handling portion of the compiler.

Work is presently proceeding on modifying the interpreter design to take advantage of the capabilities of Ada for defining abstract data types and procedures for operating on them. Additionally, an implementation of the interpreter in Modula-2 is in progress. The Modula-2 version is designed to take advantage of Modula-2's capability for defining abstract data types.

Another project is underway which combines the list-processing capabilities of LISP with Polish-postfix notation and extensibility of FORTH. Only definitions and one function require prefix notation. The language is called "Fifth," and will be the subject of further research and later publication. A prototype interpreter has been implemented in Ada, using many packages written for the FORTH interpreter. A word definition in Fifth is as follows:

```
(: word_name ( list )).
```

The concept of a store is eliminated. Instead, all data is stored as the word body associated with a name. Program and data are identical: all are represented as lists.

```
procedure EXECUTE_PRIMARY(N : in INTEGER) is
begin
  REGISTER(PC) := N + CODE;
  loop
    FETCH_INST;
    exit when OP = END_TAG;
    exit when OP = 0;
    EXECUTE;
  end loop;
end EXECUTE_PRIMARY;

procedure EXECUTE_SECONDARY(N : in INTEGER) is
  IR, WA : INTEGER;
begin
  IR := N + CODE;
  UPDATE_REG(IP, IR);
  loop
    WA := MEMORY(IR);
    exit when WA = END_TAG;
    exit when WA = 0;
    if IS_SECONDARY(MEMORY(WA + TAG)) then
      EXECUTE_SECONDARY(WA);
    elsif IS_PRIMARY(MEMORY(WA + TAG)) then
      EXECUTE_PRIMARY(WA);
    else
      raise PROBLEMS;
    end if;
    IR := REG_IS(IP);
    INCR(IR, 1);
    UPDATE_REG(IP, IR);
  end loop;
end EXECUTE_SECONDARY;

procedure W_EXECUTE is
  PTR : INTEGER;
begin
  PTR := OS_POP;
  if IS_SECONDARY(MEMORY(PTR + TAG)) then
    EXECUTE_SECONDARY(PTR);
  elsif IS_PRIMARY(MEMORY(PTR + TAG)) then
    EXECUTE_PRIMARY(PTR);
  else
    raise PROBLEMS;
  end if;
end W_EXECUTE;
```

Figure: Inner Interpreter Algorithm