# FIFTH-GENERATION FORTH

James C. Bender
Texas Instruments
Dallas, Texas

## 1. ABSTRACT

Fifth-Generation Forth is a language which combines characteristics of both Lisp and Forth. The core vocabulary includes built-in functions supporting object-oriented programming (the Simple system, developed for Prolog), and forward and backward chaining functions. Other built-in functions support list-processing, string-processing, and frames (as defined in Winston's Artificial Intelligence, 2nd Edition. Fifth-Generation Forth, then, has many attributes of an expert system building tool. An interpreter is under development for the IBM PC and compatibles.

## 2. PHILOSOPHY

Fifth-Generation Forth is a high-level language which combines a Forth-like syntax with the list notation of LISP, and with an object-oriented view. Fifth-Generation Forth uses the Simple object-oriented system as the basis for a frame-based knowledge representation system and for a production rule system. In effect, then, Fifth-Generation Forth has all the features of an expert system building tool, with the addition of a procedural, Forth-like programming language.

The fundamental components of Fifth-Generation Forth are a stack of pointers to objects (symbols, numbers, and lists), and a dictionary of symbols and values. The values in the dictionary can be either program or data: there is no distinction. The dictionary is structured so that function definitions, data, production rules, and facts are stored in separate areas. Functions are provided for defining and manipulating each type.

Despite the mixture of features, a uniform notation is used which will look very familiar to Forth programmers.

## 3. FUNCTIONS

Function names were chosen from several sources: Forth, Lisp, and Flavors. Flavors is the object-oriented programming system for the Lisp machine environment. Stack-manipulation functions use the standard Forth names while list-handling functions use the Lisp names. Built-in functions are provided for object-oriented programming, for the frame-based knowledge representation, and for the production rule system, in addition to the usual I/O, string-handling, and list processing functions.

## 4. CONTROL STRUCTURES

Besides sequential and procedural abstractions, three control structures are provided: "cond," a Lisp-like case structure; "loop," an iteration with an "exit" at an arbitrary position in the loop; and "for," a loop which is executed an integer number of iterations.

### a. Conditional (COND)

The format of the "cond" function is as follows:

```
        cond (((Condition-1)(Action-1))


                  ((Condition-i)(Action-i)))
```

Each condition is executed in turn with a flag left on the stack each time. The value of the flag must be either TRUE or FALSE, or an error has occured. When a TRUE flag has been encountered, the corresponding action is executed and control falls through to the end of the list of condition-action pairs. A default is denoted by the use of TRUE as the only item in the condition. Both TRUE and FALSE are pushed to the stack when they are executed.

b. Iteration (loop)
The format of the "loop" is as follows:

```
       ... loop ( functions ) fn-i ...
```

The list following "loop" is executed indefinitely, or until an "exit" is executed. "exit" causes control to transfer to the function after the list following "loop."

c. "for" Loop

The format of the "for" loop is very similar to that of the "loop," except that the list following the "for" is executed a definite number of times. "for" expects an integer number on the stack which specifies the number of loop iterations to be executed. The format is as follows:

```
       ... for ( functions ) ...
```

The stack is assumed here to have an integer number left on the stack.

## 5. OBJECT-ORIENTED PROGRAMMING

The Simple object-oriented programming system, as implemented in Fifth-Generation Forth, is derived from both Smalltalk and Flavors. Classes of objects, subclasses, and instances of classes can be defined. Software entities are used to correspond to physical or conceptual entities in the problem space. Generic messages are defined for classes which are inherited by instances of the classes. Messages may be sent to instances, which invoke the execution of an associated function or "method." Methods are stored in class objects.

Only five functions are necessary to use Simple: "defclass," "make_instance," "defmethod," "defvar," and "send_message."

a. "defclass" is used to define classes and subclasses. The format for "defclass" is as follows:

```
           ... 'Classname 'Superclasses defclass ...
```

If the class is a root class, with no super classes, the super class parameter must be a null list--"()" (the Lisp "NIL"). The apostrophe (') is a function (like the Lisp QUOTE) which inhibits the execution of a symbol or list which immediately follows. The result is that a pointer to the item following the quote is pushed to the stack.

b. "make_instance" is a function which defines an object which is a member of a class. The format of "make_instance" is as follows:

```
... 'Class 'Instance_name make_instance ...
```

An instance is a member of only one class, although classes can have multiple superclasses. An instance inherits methods and instance variables from the class to which it belongs. When an instance is created, methods are automatically created for all instance variables which it inherits. If an instance variable has an initial value, that value is stored as the value of the variable in the instance object.

c. "defmethod" is a function which defines a function attached to a class to perform a generic function. The format of "defmethod" is as follows:

```
... 'Function-name 'Message 'Classname defmethod ... .
```

The "Message" parameter is the generic message name which is used to invoke the execution of the method function.

d. "defvar" is used to define instance variables. Instance variable values are stored in instance objects, using methods that are created when the objects are instantiated. The format of "defvar" is as follows:

```
... 'Instance-var 'Class defvar ...
```

Using the names in this example, the method for reading the value of an instance variable looks like :Instance-var. The method for setting the value is of the form: :set-Instance-var.

e. "send_message" is a function used to pass a message and, optionally, arguments, to an instance object. Any returned value is left on the top of the stack. The format of "send_message" is as follows:

```
... '( Arg-list ) 'Message-name 'Instance-name send_message
```

The arguments must be contained in a list. This is done for ease of implementation and efficiency.

## 6. FRAME-BASED KNOWLEDGE REPRESENTATION SYSTEM

A frame-based knowledge representation system accompanies Fifth-Generation Forth. Frames are used to build a semantic network. This system allows the dynamic creation of frames, slots, facets, and values. A quadruple of a frame name, a slot name, a facet name, and a value can be thought of as a relation. A slot is an attribute of the frame. A facet is an attribute of a slot. Values are usually stored in the value facet of a slot.

There are a large number of frame-handling functions in Fifth-Generation Forth. The most often-used functions are: "fput" (to put a value into a frame-slot-facet-value relation), "fget" (to retrieve a value), and "fremove" (to remove a value). For a more complete explanation of frames see Winston and Horn's Lisp book (2nd edition).

## 7. PRODUCTION RULE SYSTEM

The production rule system included as part of Fifth-Generation Forth allows the definition of "if-then-do" rules with variables, and unstructured facts. The rules can be driven either backward or forward. Rules and facts can be added dynamically during execution. Usable systems must allow the set of rules and facts (a knowledge base) to be modified during program execution. The forward chaining inference engine requires this capability in order to work. As rules fire, the consequents of rules (the "then" part) are added to the knowledge base. Functions are provided to define rules and facts, to remove them, and to initiate the forward and backward chaining reasoning processes.

## 8. IMPLEMENTATIONS

A prototype interpreter was written to aid development of language concepts, and for an IEEE-488 control application. This interpreter did include support for the fundamental frame operations at the core of the Simple object-oriented system, but did not have built-in forward and backward chaining functions. Built-in functions were provided for IEEE-488 control.

A Fifth-Generation Forth interpreter is under development for IBM PC's and true compatibles. This interpreter will support the full definition of the language. It is likely that graphics will also be available, since the underlying implementation language provides graphics support.

## 9. CONCLUSION

Fifth-Generation Forth is a procedural language which uses a Forth-like syntax and a Lisp-like list notation. Features included in the language include the Simple object-oriented system, a frame-based knowledge representation system, and a production rule system. Fifth-Generation Forth is suitable for developing a wide variety of artificial intelligence applications, including expert systems and natural language processing. Real-time efficiency is sacrificed in the design and implementation in exchange for a powerful knowledge representation system and the facility for expressing both domain-related rules and meta-rules. The design of the language has evolved out of an earlier, high-level language implementation of Forth.