Fifth: A Forth Based Programming Environment

Cliff Click and Paul Snow
P.O. Box 10162
College Station, Texas 77840

May 17, 1986

Abstract

Forth is an excellent base for a complete programming environment. **Fifth** is a version of **Forth** with memory management, scoping, a large memory model, and a highly visual programmer interface. **Fifth** allows editing on two levels, the module level and the program level. Modules are easily modified without changing their position in the dictionary. The dictionary can be easily rearranged using the dictionary editor.

Introduction

We began experimenting with a programming environment in December of 1984. Our goal was (and is) the unobtainable, the "perfect" programming environment. Our interest came from a simple observation: the programming environment often makes a bigger difference in productivity than the language. (Why else would anyone use BASIC?) We wanted the following characteristics in our environment:

- Fast compilation. We found that we often forgot what we were doing (and why!) while waiting for compilations to complete.
- Modular code. Complex functions should be easily hidden away within routines which can subsequently be used as "black boxes".
- Efficient code. Programs produced in the "perfect" environment should be competitive in speed.
- Hidden compiles. With a fast compiler, there is no need to bother the programmer with details like invoking the compiler.

Forth was used as the system language even though neither of us had any significant **Forth** programming experience. Why? In contrast to languages like C or Pascal, **Forth** is simple to implement and modify; this let us concentrate on environment issues. (Had we used C as a base, any change could cause side effects through out the compiler; we would have spent all our time on the compiler rather than the environment.)

As we looked into **Forth**, we found it was lacking a few features that we felt were essential to the environment we wanted to build. These include:

- A large memory model.
- Scoping.
- Memory management.
- A visual programming interface.
- Source level tracing and breakpoints.

The large memory model is a need recognized by many venders of 32-bit Forth systems. Scoping is needed to reduce name collisions. Convenient memory management gives programs easy access to dynamic memory; this is a feature needed in many AI applications. **Fifth**'s programming interface also relies heavily on dynamic memory. This interface is visual, allowing the programmer to inspect and change the dictionary. It gives the programmer a better "feel" for the state of the system. This interface is enhanced by **Fifth**'s capability of selectively tracing modules at the source level.

Scoping

We had one major goal for the scoping rules in **Fifth**: to help the programmer build a more accurate, functional "model" of a program. **Forth** makes use of linear scoping; words in a **Forth** system are defined using previous words. Where as this kind of scoping is an improvement on global scoping, we wanted more flexibility in building program "models" within **Fifth**.

The IBM PC version of **Fifth** uses scoping rules similar to Pascal. Every module[1] "owns" a subdictionary. Within a subdictionary, modules may only use the previous modules. A search for a particular module ends at the first match (just like **Forth**). If the search is not satisfied in the subdictionary, a search is made starting with the owner of the subdictionary, and of the modules visible to the owner. This search continues until the last module, the root module, is reached. (The root module is the module that "owns" the dictionary.) If the module is not found, the module is undefined.

Fifth's scoping rules greatly improve the program "model." Modules are made more independent by "hiding" supporting modules in subdictionaries. A module for all appearances might seem to perform some complex function, but in reality, the function is performed by a set of modules, neatly hidden from view. This encourages small sets of code without cluttering the dictionary with words that are of little or no interest to most of the program. Faster compilation is a pleasant side effect of scoping since scoping shortens the search path.

[1]At this point we should justify the use of the term "module" in the place of "word." A **Fifth** module includes not only its own code, but also a subdictionary containing (possibly) other modules and their code and subdictionaries. A module (and its subdictionary) may be saved to (and loaded from) disk separately. Modules in **Fifth** make it easy to build and support libraries of useful routines which can be easily merged or blended in with other programs.

Fifth's Dictionary

Screens are a great idea; programs should not be written in large, linear files. But dealing with numbered screens can be painful too. **Fifth** gives each module its own "screen," which **Fifth** calls the module's text. A module's text contains only the source code for that particular module. The text for a module is not restricted to a particular number of lines, and is compressed to reduce memory requirements. But the biggest departure from screens is in where **Fifth** keeps the text for a module; it is kept in the dictionary with the execution code for the module. This means that a module owns its own source code as well as its execution code *in the dictionary*. Furthermore, the dictionary is not the current state of a program; it *is* the program.

Memory management becomes necessary for a **Fifth** system because modules may grow or shrink in size as they are modified. **Fifth** keeps modules in in a heap rather than a stack, and module references are indirect so the system can move them. **Fifth** lets the programmer manipulate modules in the dictionary while limiting side effects to other modules in the dictionary.

Despite all the differences between the dictionaries of **Fifth** and **Forth**, there is little difference in the philosophy. A module's behavior is still similar to a word in **Forth**, and **Fifth** encourages the programmer to build his programs from small, easily tested modules. **Fifth** simply extends the **Forth** philosophy by simplifying the interface. This interface is the subject of the next section.

Fifth's Dictionary Editor

The dictionary editor is the key to **Fifth**'s programming environment. It is the program that allows the programmer to freely inspect and modify individual modules and/or their position in the dictionary. The programmer can concentrate on a module, debugging it where it is in the program, with changes coming into effect immediately and automatically. Or the programmer can concentrate on the organization of the program, moving related modules to logical positions, deleting unneeded modules, or adding new modules. In **Fifth** the dictionary takes an active part in defining the program, directly expressing the relationships between modules.

A programming environment should have some method of maintaining libraries of useful functions. In **Fifth**, libraries of routines are maintained using the dictionary editor. It enables the programmer to save out a module (and its subdictionary) independent of the rest of the dictionary. Conversely, modules may be loaded back into the dictionary at any point. In this way the dictionary editor supports easy maintenance of libraries of routines which in turn increases reusability of code.

Automatic Compiler

Compilation is an integral part of **Forth**, and is done when a screen is loaded or code is typed. In **Fifth**, the text for each module is loaded into the dictionary *without* compiling it. In addition to the text, each module is given a program stub. Executing a module executes this program stub which in turn invokes the compiler on the module. (This program stub is given to all modules "marked" as uncompiled.) If the compilation is successful, the generated code replaces the program stub, and is executed. The first execution of a module invokes the compiler first, then later execution of the module executes the module's code directly. The programmer never has to invoke the compiler.

Fifth uses a number of strategies to reduce the compilation time of a program. One strategy is analogous to lazy evaluation, where compilations are delayed until a module is actually executed (lazy compilation?). Another is the observation that compiles normally follow editing; **Fifth** employs an incremental compilation scheme where the compiler is called automatically upon leaving the editor. This compilation is normally restricted to the modified module; modifications are reflected almost instantly, even in large programs. There are operations that cause **Fifth** to mark a number of modules as uncompiled, causing noticeable compile times. These include inserting, deleting, or moving a module.

The programmer is not restricted to **Fifth**'s automatic compilation; the programmer can force compiles. This can be done from the dictionary editor, or from within a program. A module can actually force the compilation of another module. **Fifth** modules can use conditional compilation to increase execution speed, and recompile a module if conditions change.

Summary

Fifth encourages the isolation of a module from the rest of the program by placing supporting modules in subdictionaries. The programmer can easily inspect and change the relationships between modules using the dictionary editor. This improves the maintenance and flexibility of a program. The scoping rules and dictionary editor team up to eliminate the need for vocabularies and special schemes for handling libraries of routines. **Fifth**'s automatic compilation of modules greatly reduces compile times, and reduces the programmer's burden of maintaining the state of the program.

An Implementation of Fifth

We have developed a version of **Fifth** that runs on the IBM PC. It uses 32b arithmetic, a large memory model, floating point, graphics, source level tracing, and MS-DOS files. The text editor, dictionary editor, and primitives are memory resident. "Online" help is available for all primitives. This version is a Freeware program.

We are also have an Amiga version which includes improvements in the flexibility of **Fifth**. A generic version written in C is in the works to get **Fifth** up and running on any system with a good C compiler (e.g. UNIX).