A THREADED FP INTERPRETER/COMPILER

Surapol Pungsornruk

March 1986
CSD-860061

ACKNOWLEDGEMENTS

Thanks are due to Drs. Dan Berry, Milos Ercegovac, and Tomas Lang for helping to improve this report. Work by Jose Arabe and John Worley helps shape the *FP* execution environment implemented in this project. This work has been supported in part by ONR Contract N00014-83-0493, and in part by my wife, Kritaya Kongsuwan.

ABSTRACT

A Threaded FP Interpreter/Compiler

by

Surapol Pungsornruk

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor Milos Ercegovac, Chair

In this work, the technique of threaded interpretation is used to compile *FP* programs. By organizing the lexicon, used in threaded interpreters to hold function definitions, as a jump table, deficiencies encountered in conventional threaded interpreters are overcome. Combining this approach with a simple but efficient garbage collection strategy results in a fast implementation of a *FP* interpreter/compiler.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Functional programming is generally taken to mean a programming system in which functions are defined to be the building blocks of programs. Such programs differ from conventional programs in several respects.

1. Expressions exhibit applicative structure. Operator part denoting function is seen clearly to act on operand part denoting value.

2. Computation proceeds by stages, each stage transforms its input producing value for the next stage. This is commonly known as locality of effect.

3. Objects are referentially transparent. This means that an expression occurring twice in the same context denotes the same value at both occurrences.

4. There is no assignment statement.

These features allow some algorithms to be expressed clearly and concisely. In particular, since FP programs explicitly show data transformation as part of the process of computation, parallelism in these programs can be identified more clearly and exploited with greater ease.

1.1 FP Systems

Backus[1], in his seminal paper on a functional style of programming, proposes a hierarchy of alternative systems.

At the lowest level is the *FP* system in which the concepts of variable, explicit state representation and history sensitivity have been abandoned. Furthermore, all control constructs are replaced by combining operators that manipulate functions directly. These operators, called functional forms, are higher level functions; they take functions as arguments and return functions as results. A *FP* program is then one or more function definitions, using functional forms and primitives.

Next up in the hierarchy is the *Formal FP* system. This reintroduces the concept of history sensitivity by means of named cells. Functions are viewed as objects stored in cells; their meanings can be retrieved from cells by naming. Function redefinition can be accomplished by writing over existing meanings in cells. The system also allows expressions to be written in which objects denoting functions are applied to objects denoting value[1]. As a result of these capabilities, it is possible to transform programs, *i.e.* create new primitives or functional forms[2].

The last system described by Backus is the *Applicative State Transition* system. This is the type of system that is seen by Backus as an alternative to von Neumann system. Such a system employs an applicative subsystem (such as *FFP*) as computation model. It changes state by transforming its set of defined functions and known values. State transition takes place as a result and at the end of computation by the applicative subsystem. By virtue of *FP* semantics, there is no side-effect.

[1]In *FP*, application is a system operation and not part of the language.

[2]It is important to note the difference between user-definable functions, as commonly found in a *FP* system, and creating new primitives or functional forms in a *FFP* system. In *FP*, it is never possible to create new functions as a result of running a program, while in *FFP*, it is possible to do so.

1.2 Project Goal

Various implementations of *FP* interpreters have been done over the years. These are either Lisp-based[2, 3] or written in *C* [4]. Lisp-based systems incur speed penalty because of the underlying Lisp implementation. The interpreter in *C* [4] is not interactive and has different syntax from that of more widely-used Lisp-based ones. There is but one known attempt to compile *FP*[5]. Lu[5] describes a translator to compile *FP* programs into an intermediate form, which is then rendered into *C*, and finally compiled into machine code. This approach has the advantage that the program can be optimized easily while in intermediate form. The three-pass translation process, however, is cumbersome and not suitable for use in interactive development of *FP* programs.

The goal of this project is to explore an efficient way of executing *FP* programs on currently available computers, so that use of *FP* is not hampered by slow response.

While compilation is commonly used to speed up the interpretive process, it is limited by the lack of variables in *FP* programs. An imperative high level language program contains data definitions, instances of defined data, and code sequence to manipulate them. Data definitions are translated into layout in machine words; data instances are mapped into memory locations so that references to them can be generated in compiled code. Without variables, *FP* programs cannot be compiled in the same way. To be able to deal with data which can be of any type or structure, and not known at function compile time, interpretation is necessary. To achieve speed, the overhead of interpretation must be reduced. This is achieved by applying the technique of threaded interpretation to compile *FP* programs.

1.3 Threaded Interpretation

Most language interpreters work in two phases. The first phase analyzes the source language to produce an intermediate form such as a parse tree; the second phase then interprets the intermediate form at run time. A threaded interpreter produces a completely analyzed internal form, eliminating further analysis at run time[6].

Specifically, threaded interpretation is a technique of compiling an input expression into pointers to previously compiled components of that expression. By assuming the existence of a set of defined primitives, expression interpretation becomes a process of chasing pointers, leading, in the end, to execution of primitives. It is obvious that the speed of such an interpreter depends greatly on the speed of mechanism used to chase pointers. The most common use of threaded interpretation now is found in *Forth* interpreters[6].

1.4 Forth

Forth is a compact language originally used for real-time applications such as instrument control, image processing, and graphical display on microcomputer systems. It uses a stack-based computation model, with a reverse polish notation heavily emphasizing this.

A *Forth* program consists of *words* which are either primitives or user-defined. A primitive specifies an atomic operation, such as *ADD* or *POP-STACK*. A word is defined by a list of words (already defined), encapsulating a sequence of operations. Words and their attributes are kept in a database, called a *lexicon* or *dictionary*.

The meaning of a word is found in the lexicon. If it is a primitive, its meaning is specified by a routine. If it is a defined word, its meaning is specified by a list of lexicon pointers which point to words forming the definition for that entry. Computation proceeds by recursively retrieving meanings of words from the lexicon until routines are found and executed. A control stack is used to store return addresses.

Compiling a word is a very simple process, since a word can only be defined by words already defined. As each word is defined, its entry is appended to the end of the lexicon. Multiple definitions of a word are allowed to exist in the lexicon. During compilation, the lexicon is searched from back to front. Hence it is possible to have words defined which use different versions of the same word.

1.5 Report Outline

The rest of the report is organized as follows: chapter 2 introduces the design of a threaded *FP* interpreter/compiler; chapter 3 discusses its implementation and aspects of memory management, and chapter 4 deals with performance evaluation of this interpreter.

CHAPTER 2
SYSTEM DESIGN

Several objectives have to be met by a successful design. First of all, while speed is of paramount concern, it is to be achieved without sacrificing the convenience of interpretation. Secondly, the system is to run a large portion of existing *FP* programs without modification. This means that the syntax of an existing *FP* interpreter has to be adopted. Lastly, the resulting implementation should be easy to maintain, expand or modify. In what follows, the system is first described; design issues are then raised and their solutions discussed.

2.1 System Organization

Figure 2.1 shows an overview of the system. By using a syntax-driven translation process, input is broken into tokens and fed to a parser. If a function is being defined, the parser builds a parse tree; otherwise the parser passes the tokens to a system function dispatcher. In the case of function definition, the parser calls the compiler after the parse tree is built. The compiler allocates a lexicon entry for the function and passes the parse tree to a code generator which traverses the parse tree producing machine codes, generating calls for primitives or functions, including itself.

2.2 The Lexicon

Defined functions are kept in a database called lexicon. Each entry in the lexicon needs to contain:

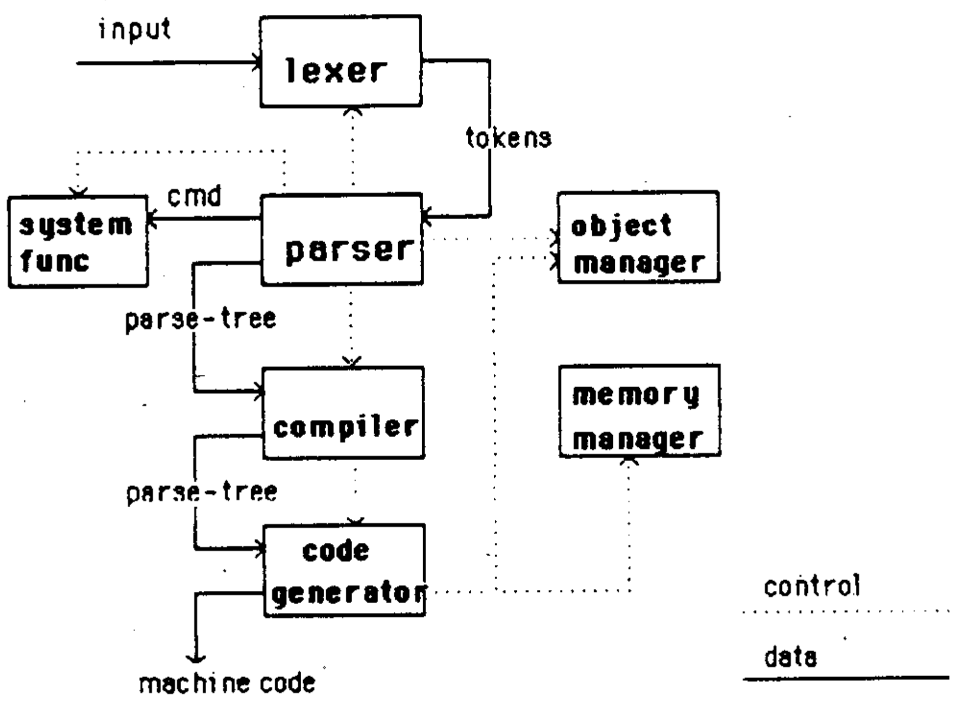


Figure 2.1 System Overview

1. a unique function name, allowing function reference, and

2. a code field, containing code or pointers to code.

The organization and management of the lexicon determine how function references are resolved during compilation, as will be discussed below.

Since function names of arbitrary length are allowed, a pointer is kept in the lexicon instead of the actual character string. Since the size of compiled code for each function is subject to change, the code field is separated from the lexicon and replaced by a code pointer. This necessitates the management of variable-sized code blocks, requiring an extra field in each lexicon entry for code block size.

2.3 Compiling and Interpreting

To achieve speed and flexibility, a hybrid approach of compilation and interpretation is used. The basic idea is to use threaded interpretation; but instead of compiling an expression into pointers, it is compiled into subroutine calls. This allows pointer-chasing to be done at the speed of a *call subroutine* machine instruction, reducing substantially the overhead incurred as compared to that of an address interpreter commonly used in a threaded interpreter.

As in other compiling systems, the input expression is discarded after compilation. Thus, at the expense of not keeping input expressions, syntactic analysis is done only once for each function defined.

To allow for interpretation, the parser is designed to allow legal *FP* expressions to be applied to data objects. In this case, the *FP* expression is compiled for a function with a reserved name. To avoid over-writing existing functions, the reserved name is chosen so that it is not a legal name for a *FP* function. The result of interpretation is obtained by applying this function to the given data object.

2.4 Resolving Function Reference

There are, however, problems with threaded interpretation. Conventional *Forth* interpreters have the following restrictions.

i. The order in which functions are defined is important. A function may call only functions already defined.

ii. Redefinition of a function has no effect on functions already defined using that function.

These two problems are related to how function reference is resolved during compilation. Enforcing (i) avoids having to resolve function addresses at run time, but does not solve the problem of redefinition. *Forth* systems address this problem by allowing users to issue a command and wipe the lexicon clean. This allows the lexicon to be purged of old versions of functions. Since function redefinition commonly occurs during program development and debugging, after purging, a working program can be compiled into an empty lexicon, free of any stale definition. This is inadequate because an interpreter should be able to relieve users of such responsibility. Besides, for interactive use, it is not reasonable to force users to adopt a bottom-up approach in program development, requiring callee functions to be defined before caller functions.

There are two ways of solving these two problems. Function references can be resolved by users invoking a system facility prior to running. This is identical to the conventional approach of a "link" command for compiled languages such as *Fortran* or *C*. The user must bear the burden of remembering which functions to relink after each redefinition.

To make relink transparent to users, it can be done at run time or compile time by the system. Run-time relinking is not really practical. Depending on design, as many as all functions called by the invoked function need to be relinked. To automatically relink all caller functions when a callee function is redefined is more reasonable. However, it is not clear how this can be done without messy housekeeping of function names and addresses. Moreover, as more functions are defined, the linking process will get more expensive.

To allow function reference to be resolved at compile time, and obviate the need to relink, function addresses are made invariant. This is accomplished as follows. Each user-defined function is assigned an entry in the lexicon. The location of the entry in the lexicon for each function is permanent. By keeping a pointer in the code field of the entry, function code can be accessed via the lexicon. Hence, function references can be resolved at compile time. Redefinition is handled by setting the code pointer to point to the current version of code. There is no need to relink, because lexicon entries for functions always point to the latest version of code.

During compilation, lexicon entries are constructed for functions which do not have them. There are two ways in which this can happen.

1. A function is being defined for the first time.

2. An undefined (new) function is used in the definition of another function.

For either case, a full entry is constructed. But for the second case, the code field of the entry is set to a routine which warns the user of undefined function. As a result, functions which are called but not defined, issue warnings when they are executed.

2.5 Execution Environment

All *FP* primitives and functions take one operand and produce one result. This makes the use of a stack a convenient way to pass arguments and return results. Both primitives and functions take their operand from and leave their result on top of a data stack. There is no need to keep track of how many arguments are needed by each function, reducing the size and complexity of the lexicon.

2.6 Memory System

There are two separate requirements for memory: to hold defined functions, and run-time objects. Defined functions are stored as entries in a data structure called lexicon, with an auxiliary area for compiled code. Run-time data objects are stored in a data stack and a heap.

There are two ways to allocate memory for these data structures: statically by declaring structure size when the system is built (compiled), or dynamically at run time, requesting services from the operating system on host machine. While relying on operating system to dynamically allocate storage would relieve the system of declared limits in data structure size, it is not done for reasons outlined below.

The available software library on UNIX to do dynamic allocation is not efficient in a virtual memory system[7]. As stated in the *BUGS* section of the *malloc* manual page[7], where a large number of small blocks are managed, each allocation can cause all allocated and freed blocks to be referenced, creating a large number of page faults. Since the target machine is a VAX, which is a virtual memory machine, the system page fault handling mechanism can be used directly and efficiently in the following scheme.

All data structures are implemented as arrays of cells with declared limits in the virtual address space. Each area is then managed by dynamic allocation and garbage collection at function compile and run time. Memory pages for these are brought into physical memory by the host memory management system, only when those locations are addressed. Fortunately, the size of the executable file for the interpreter is not affected by the size of these declarations. Although the interpreter is now subject to limits in array sizes, it gains in independence. Memory management of the arrays can be implemented as a self-contained module, quite separate from any reliance on operating system.

2.7 Expression Syntax

To allow a large portion of existing *FP* programs to run without modification, it is decided to use *Berkeley FP* syntax. Besides being the most widely available interpreter, it is probably also the most completely documented. Users are referred to *Berkeley FP User's Manual* [2] for a complete account of expression notation, primitives and functional forms. Owing to the way this interpreter is implemented, it should be straight forward to modify it to accept other syntax.

CHAPTER 3
IMPLEMENTATION

The system is implemented in *C,* with some assembly routines. This is reasonable, since *C* is considered suitable for system programming and speed is one of the primary goals of this project. *C* is also better supported on UNIX than other procedure-based high level languages. Tools such as *lex* and *yacc* are added bonuses. The target machine is a VAX computer running UNIX. The system is not portable since compiled code for *FP* functions is generated to execute on the VAX.

Implementation of the system can be discussed under two broad headings: function compilation and function application. In this chapter, representation and storage of data objects are first described. Compilation is then treated in some detail, followed by a description of the run-time environment, including memory management.

3.1 Data Objects

In *FP,* there are two types of objects: atom and sequence. There are two places in the system where data objects are stored: data stack and heap. Both the data stack and heap are implemented as arrays of cells.

3.1.1 Stack Objects

A stack cell consists of two fields, a tag field and a value field. The tag can take different values to identify the type of object represented. To enumerate, the following types of objects are represented: integer, floating-point number, boolean, *bottom,* null sequence, character string, and sequence pointer.

In the case of integer, floating-point number, boolean, or sequence pointer, the value of the object is found in the value field. In the case of *bottom* or null sequence, the value field is not used. For character string, the value field is a string pointer. Atoms are stored on data stack, one per cell; sequences are always stored as pointers, pointing to actual sequences in the heap.

3.1.2 Heap Objects

A cell in the heap consists of four fields: a tag field, a value field, a link field and a structure field. The first two fields are identical to those of a stack cell. The link field is used to hold a cell pointer, and is used to represent sequence, like a list is represented in Lisp, by setting the link field to point to the next cell in the sequence. The structure field identifies if the cell is part of a sequence, and allows atoms to be stored in the heap. This is necessary since the tag field only identifies the data type of the value field. Sequences can also be replicated in the heap by storing a sequence pointer in a cell, with the structure type indicating the cell not to be part of a sequence. A tag value, not used in stack cells, identifies if a heap cell is in use or free. This is used in heap management as will be described in section *3.4.2*.

3.2 Function Compilation

Compilation is a two-stage process. Given a function definition, a parse tree is first built, code is then generated for each node in the tree. The compiling routines deal with several data structures which are briefly described below. The process of building a parse tree and generating code is also discussed.

3.2.1 Compiling Environment: lexicon and code area

The lexicon is the database holding all user-defined functions. It is implemented as a jump table. Each entry in the jump table consist of four fields: the first field is the machine code for an absolute jump, the second field is the address of compiled code for that entry, the third field is a character pointer, pointing to the function name, and the fourth field is the length in bytes of compiled code.

The system maintains a contiguous block of memory, called *code area*, to hold compiled code for functions in the lexicon. Each piece of compiled code for a function ends with a *return from subroutine* instruction. Since the VAX architecture does not enforce word alignment for instructions, code pieces for functions are concatenated byte-by-byte without holes. Since the position of a function in the jump table is never changed, calls to functions are generated to their addresses in the jump table. As a result, the compiled code of a function in the code area can be moved and still remains accessible to all functions.

3.2.2 Primitives and Functional Forms

Routines for primitives and functional forms are written in *C* as part of the system implementation of a *FP* machine. For primitives, a system table of pairs of primitive name and routine pointer is maintained. Entries in this table are resolved at system compile, link time, and are available to the code generator.

3.2.3 Parsing

The front end of the system, the lexer and parser are implemented using utilities *lex* and *yacc*, respectively. The parser builds a parse tree for each function defined.

3.2.3.1 Parse tree

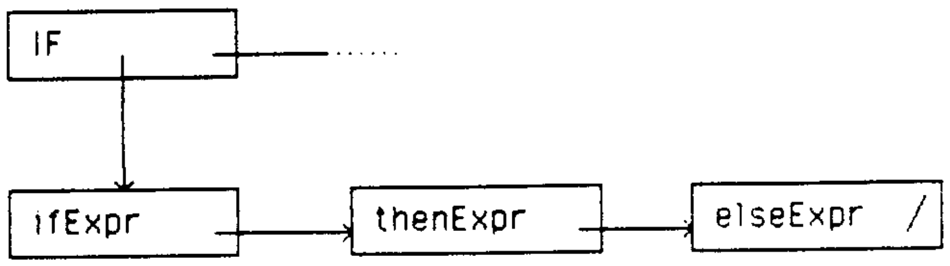
Parse trees are built out of a data structure called *node.* A node consists of three fields: a token field, a value field, and a link field. The token field identifies what type of *FP* language construct is being represented. The value field takes on different meanings depending on token type. The link field is used to hold a node pointer. The parse tree for each *FP* construct is shown in figure 3.1.

Parse trees are built using a left-most-child, right-sibling representation. Each *FP* construct can be represented as a parse tree. The token field of the root node identifies if the node is a primitive, one of several functional forms, or a function. For primitives and functions, the tree consists of a single node, with the value field either identifying the primitive represented, or holding a pointer to the function name. For functional forms, the tree usually consists of more than one node. The token field of the root node identifies the type of functional form, the value field assumes the role of left-child pointer, pointing to other nodes or objects. In either case, the link field of the root node is used to combine the tree with other parse trees to form a bigger tree. An example of a parse tree for a *FP* expression is given in figure 3.2.

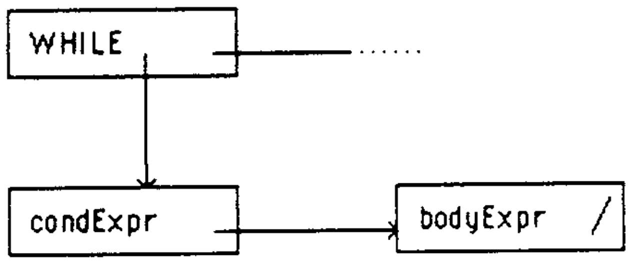
3.2.4 Code Generation

To generate code for each parse tree, the code generator traverses the tree calling routines to write out machine code to a scratch pad area in memory. When compilation is complete, the length of code generated is noted in the lexicon entry for that function, a code block of the right length is allocated from the code area, and lastly, the code is copied from scratch area to the code block.

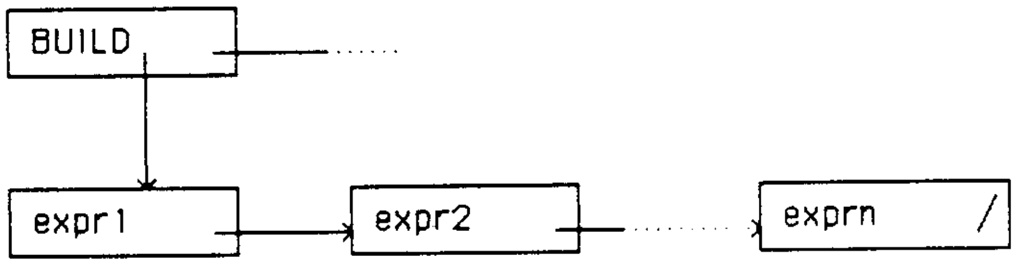
(ifExpr -> thenExpr; elseExpr)



(while condExpr bodyExpr)



[expr1, expr2, ..., exprn]



%x

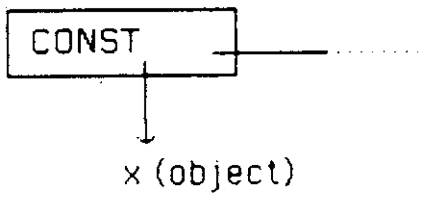


Figure 3.1a Parse Trees, part 1

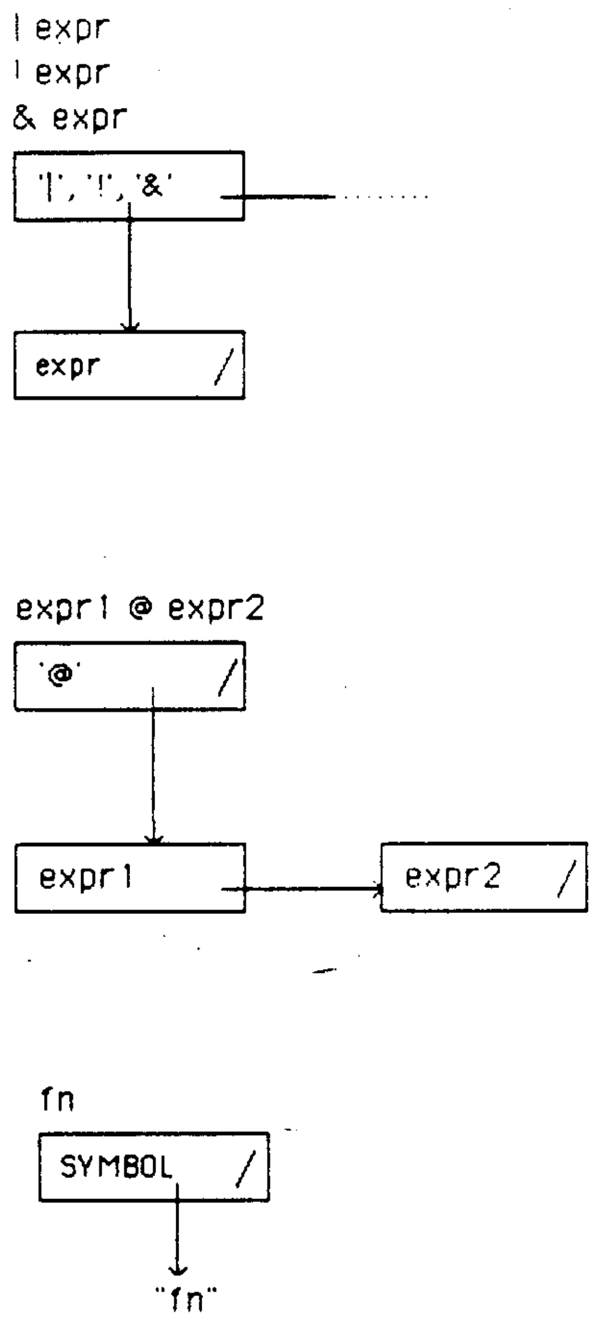


Figure 3.1b Parse Trees, part 2

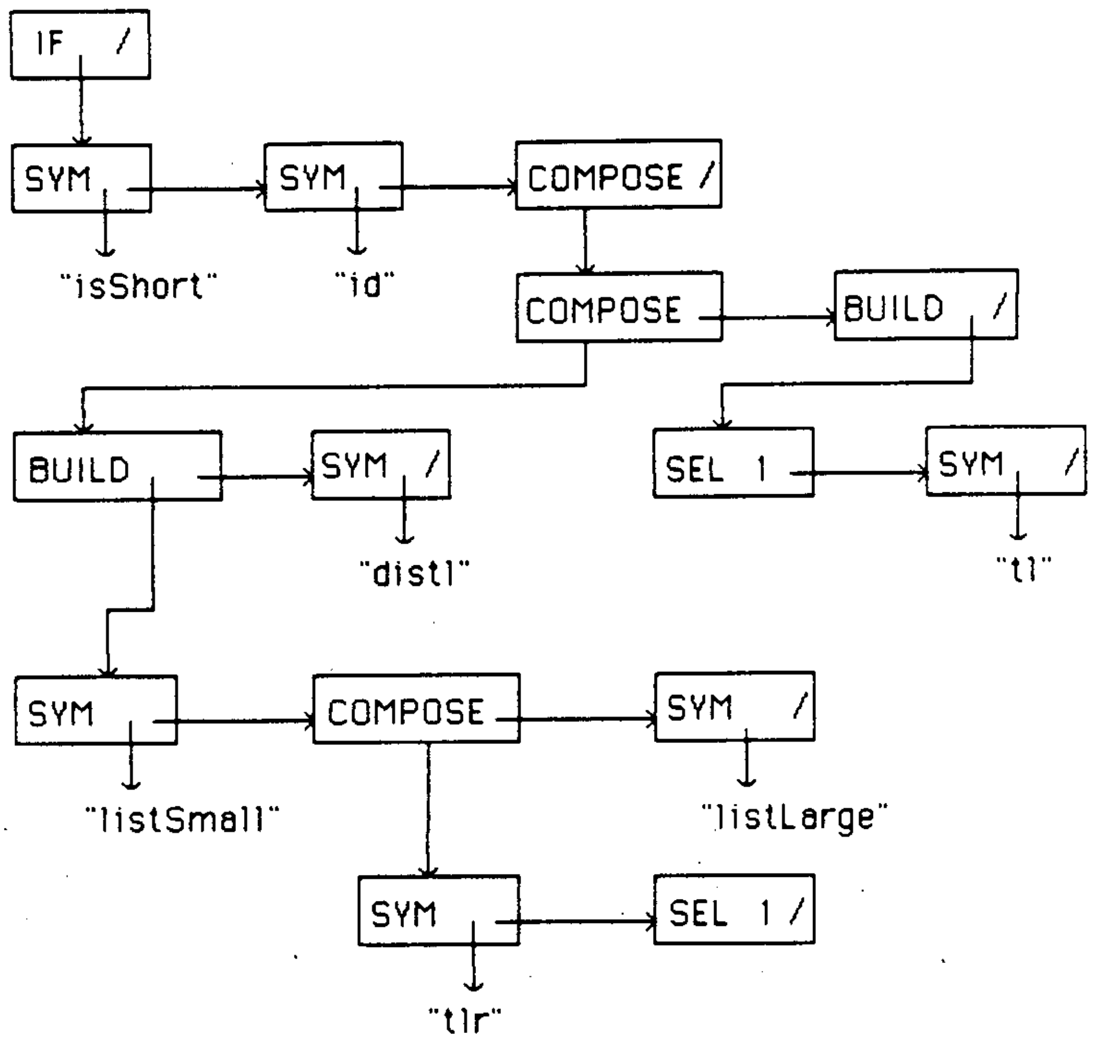


Figure 3.2 Parse Tree for FP Expression Below

```
(isShort -> id ;
        [ listSmall, tlr @ 1, listLarge ]
        @ distl
        @ [ 1, tl ] )
```

Since code pieces are moved about, care must be taken to ensure that relocatable code is generated. This is easy to do, since each piece of code is completely self-contained. The compiled code for a function typically contains calls to primitives or other functions. If the function is defined using functional forms, there are also some instructions for control flow. Instructions for calls are generated using absolute addressing mode. This works since primitives are not moved, and references to other functions are made via lexicon entries which are never moved. Position dependence in compiled code is only found in branching instructions used to transfer control to locations within a code piece. By using relative branch instructions, position independent code is generated.

The code generated for primitives and functions is simple, being just a call to the appropriate routine. For functional forms, more work is needed. The following section briefly describes how code is generated for each functional form.

3.2.4.1 Generating Code for Functional Forms

The *COMPOSE* functional form expresses sequential control flow and is dealt with by the order in which the parse tree is traversed and code produced.

The *CONSTRUCTION* or *BUILD* functional form is treated by traversing the expression list forming the argument of the functional form. Code is first produced to start building a data sequence. For each *FP* expression in the expression list, code is generated to duplicate the data object on top of the stack (TOS), code is produced for that *FP* expression, and code is also produced to append the TOS to the new sequence. After the expression list has been traversed, code is produced to terminate the new sequence and replace TOS by it.

The *INSERT* functional forms are handled by generating a relative jump over the code produced for the argument of the *INSERT* which is a *FP* expression, to a *call procedure* instruction, which transfers control to a run-time routine to do the work done by *INSERT*. There is one complication. The run-time routine needs the address of the compiled code for the argument of the *INSERT*. By changing the *relative jump* instruction to a *jump subroutine* instruction, the return address pushed on the machine stack is the address of the compiled code.

The *APPLY-TO-ALL* functional form is done by producing code to form a loop. The body of the loop consists of code to move successive elements of the data sequence being worked on to TOS, and the compiled code for the argument of the functional form. The loop is bracketed by code to form a sequence using objects from TOS.

A more detailed example is given below for the *IF* construct. For clarity, pseudo-code is used as output of the compiler in place of VAX machine code.

The *IF* functional form when applied to object *x*, is expressed in *FP* as

```
(ifExpr -> thenExpr; elseExpr) : x
```

and has the following semantics:

```
if (ifExpr : x) is true then evaluate (thenExpr : x)
if (ifExpr : x) is false then evaluate (elseExpr : x)
otherwise return (bottom)
```

The code generated is shown below. It provides for all possible values generated by evaluating ifExpr : x.

These are boolean values *T*, and *F*, and the atom denoting the value of an undefined operation, *bottom*.

```
DUPLICATE TOS (TOS = top of stack)
code for ifExpr
TEST TOS
IF TRUE BRANCH TO thenExpr
IF NOT BOTTOM BRANCH TO elseExpr
RETURN
code for thenExpr
BRANCH OVER elseExpr
code for elseExpr
```

From above, the following observations can be made.

1. Mutual recursion exists between the code generator and each routine it calls to produce code for a functional form. This greatly simplifies the code generation process.

2. By knowing what functional form it is dealing with, the code generator is able to generate code to manage the data stack.

3. Run-time routines are necessary to support the execution of *FP* primitives.

Run-time routines are needed to manipulate stack objects, and to build and dissect heap objects. The collection of these routines and associated data structures form a *FP* machine.

Most run-time routines perform simple tasks, and are written in assembly code in an effort to achieve good performance. They could just as well be written in *C*, but procedure and function calls in *C* are translated to the more expensive *call procedure* instruction which builds a frame on machine stack. For routines which are short and called frequently, this is an important factor in execution speed. By writing in assembly, a *jump subroutine* instruction can be used instead.

To support execution of complex functional forms such as *TREE-INSERT*, or *RIGHT-INSERT*, the run-time routine that does the bulk of the work is written in *C*. The rationale here is that for a lengthy routine, procedure call overhead is not as important as in short routines which are called far more frequently. A slower but working *C* routine is far better than a fast but possibly buggy assembly routine.

3.3 Function Application

Evaluation is invoked by applying a primitive or function to an object. The parser calls a routine to build the object, and passes control to a system routine to handle invocation. The routine pushes the object onto data stack, checks to see if a user-defined function or a primitive is invoked, and calls the appropriate *FP* routine to transform the object on top of stack. A printing routine is then called to print out the object left on top of the stack.

There is need to distinguish between defined functions and primitives because the transfer address of the former is in the lexicon, while that of the latter is in a system table. The calling protocol of the two types of routines are also different. User defined functions need to be called with a *jump subroutine* instruction; primitives are called with a *call procedure* instruction. The difference arises because primitive routines are written in *C*, which uses a *call procedure* instruction. For user-defined function, code is generated in the simplest fashion possible. There being no need to pass arguments, a *jump subroutine* instruction suffices and is faster.

3.3.1 Environment: Stack and Heap

FP functions are evaluated in an environment with a stack and a heap.

3.3.1.1 Data Stack

The data stack is the central stage on which primitives and functions do their things, one after the other. When all is done, what is left is the result.

The stack holds atoms and sequence pointers. At the time of invocation, there is only one object on the stack; at the end of invocation there is also only one object left on the stack. In between, the stack may grow or shrink, but never becomes empty.

The depth of the stack can be estimated using the following argument. By the nature of *FP*, each function or primitive consumes one object when it starts, and produces one when it finishes. Objects on the stack are saved only when a functional form is involved. For all functional forms, there is no need to save anything beyond the object on top of the stack when that functional form is entered. Hence the stack will grow linearly with the nesting of functional forms. For recursive functions, defined with functional forms, the stack will grow linearly with the length of input data sequence.

3.3.1.2 Data Heap

The heap is used to hold objects during computation. Its management is discussed in the next section. Computation is started by user requesting a function to be applied to a data object. The input data object is built and put on top of the data stack. The specified function is then invoked. The result is left on the data stack and is printed. In between such top level applications, the heap is theoretically free. In practice, this is true except for one implementation intrusion.

FP provides a *CONSTANT* construct to replace the object applied with the object specified. The specified object is constructed and kept in the heap. This object must be preserved across top level function applications. Fortunately, since the heap is free during function definition, the compiled object can be built and kept at the beginning of the heap. A heap pointer can be maintained by the compiler and used by the heap manager to keep track of this moving boundary as permanent objects are stored in one end of the heap. Permanent objects are not interfered with during function application as a result of the policy not to destroy run-time objects.

Run-time objects are created by a mix of replication and sharing to serve the primary goal of not destroying objects in the heap, while being speedy in creating new ones. For example, to duplicate the object at the top of the data stack, the stack cell is copied to the next cell and the stack pointer updated.

During function application, the original object in the heap is not tampered with, since there may still be pointers on the stack pointing to it. Instead, new sequences are created by copying atoms and sequence pointers. Hence, sub-sequences are always shared. This sharing is guaranteed safe, since no sequence is ever destroyed or modified.

3.4 Memory Management

There are up to three memory structures that need to be managed. These are the lexicon, the code area and the heap. The lexicon needs little or no management, since entries in it are not removed, and a function, identified by its name, once allocated an entry in the lexicon, always uses the same entry. Management of the code area and the heap are discussed below.

3.4.1 Code Area Management

The code area is a contiguous block of memory, used to hold compiled code for user-defined functions. As functions are defined, compiled code pieces are concatenated one after the other. The need for management arises from two sources: function redefinition and interpretation of *FP* expressions. Both cause the code area to be fragmented.

It is easy to see how redefinition fragments the code area, since a new block has to be allocated to hold the new code, while the old block becomes free. Interpretation has the same effect, since it is implemented by compiling a reserved function. Successive interpretations is akin to redefinition of this reserved function.

Management of code area is done by keeping a list of free blocks. This list is implemented as an array of pointers. As a block is released, its pointer is stored in the list. When a block needs to be allocated, the list is searched from back to front; allocation is done on a first-fit basis. The code area is compacted when the free-block array is full.

Compaction takes advantage of the fact that each code block is pointed by only one pointer from an entry in the lexicon, and is done as follows. Functions in the lexicon are sorted by their compiled code addresses. The lowest address is compared to a target address which is initially set to the beginning address of the code area. The code bearing that address is moved to the target address, if it is not already there. (Moving is easy, since code lengths are known.) The target address of the *absolute jump* instruction in the lexicon entry is then updated.

This process is repeated for the next lowest code address, with the target address set to the end of the previous piece of code. By stepping through the sorted list of code addresses this way, the code area is compacted. The free list is then initialized to point to the one contiguous free block.

Compaction is only done at function compile time, and has no impact on execution speed of function application.

3.4.2 Heap Management

The heap is implemented as an array of cells. As stated above, the heap is free at the beginning of each top level function application. Thus, cells in the heap are allocated sequentially. There is no need to manage the heap until cells are exhausted during computation. When this happens, garbage collection is initiated.

Garbage collection is done by a simple marking algorithm. All cells in the heap are first marked free. Then by following sequence pointers on the data stack, cells in each sequence still needed for later computation are followed and marked as in use.

Marking is done in a depth first fashion. Since sequences are shared, a sequence that is found to be already marked as in use is not marked again. This avoids marking shared sequences more than once. The collection algorithm is O(n), since each sequence marked in use is visited only once.

Once the heap has been garbage collected, cell allocation is then done on the basis of examining tags to locate free cells. The cost of examining tag is paid only when free cells are scattered in the heap.

CHAPTER 4

PERFORMANCE EVALUATION and DISCUSSION

In this chapter, the effects of architectural constraints on performance are briefly discussed. The influence of heap size on program execution and memory management are then examined in some detail. Following this, the threaded interpreter is benchmarked using four programs and compared against *Berkeley FP*, which can execute programs in both interpretive and compiled modes. Lastly, possible enhancements of the threaded interpreter are outlined.

4.1 Constraints on Performance

While a compiled *FP* program generally runs faster than an interpreted one, it cannot escape from constraints imposed by the underlying system architecture, which does not provide efficient support for facilities needed for the execution of functional programs. In this section, some of these limitations are briefly discussed.

4.1.1 Object Typing

Like Lisp, *FP* needs a run-time typing system. This means that at run time, it must be able to determine the type of an object and take various actions depending on that type. The determination of object type is a substantial part of routines in the threaded interpreter, implementing *FP* primitives. For example, to perform the primitive *ADD*, the operand must be checked to be a sequence of length two. The elements in the sequence then is tested to see if they are integers, floating-point numbers, or other types not valid for *ADD*. If the elements are of valid but different types, type coercion must be performed. Thus, a simple *ADD*, needs ten lines of *C* code to implement.

Type information not only has to be checked at run time, it must be encoded in the data structures. In the threaded interpreter, tag fields are provided in cells in both the data stack and the heap. The *C* compiler on the UNIX host, compiles a *short* integer used to encode a tag into 16 bits. Hence, the tag field is thirty-three percent in area of a stack cell, and twenty percent of a heap cell. These are the prices in time and space for run-time typing.

4.1.2 Function Call and Return

The performance of function call and return is important. For the VAX architecture, there are basically two ways to call a subroutine, a *jump subroutine* and a *call procedure* instruction. The *C* compiler running on UNIX, compiles function calls using the *call procedure* instruction, which builds a frame on the machine stack to hold saved registers, return address, and other control information. The frame is at least five 32-bit long words; it could be as big as fifteen long words[8]. For systems with a write-through cache and a memory access time in the order of micro-seconds such as the VAX-11/750, such a call is an order of magnitude slower than a *jump subroutine* instruction, which merely pushes the return address on the stack.

4.1.3 Parameter and Variable Storage

A related issue is how parameters are passed to functions and how variables are stored. The *C* compiler used passes parameters from a caller to a callee by pushing them on the machine stack. The callee may then assign parameters to registers or keep them on stack. The trade-off here is between the cost of accessing stack and that of saving registers used to hold parameters. This is exactly the same consideration required in allocating storage for local variables. Local variables can be assigned to registers or stored on stack. The former method causes a bigger frame to be built on entry to the function, since more registers need to be saved. The latter method causes more memory traffic, especially when variables are written. To obtain optimal performance, detailed analysis of parameter and variable usage in the function would have to be done, before allocating storage. This optimization is not performed by the compiler used in this project.

The picture is different for global variables, since they cannot be assigned to registers and kept there across routines. This proves to be a stumbling block in implementing *FP* machine operations. Ideally, frequently used global variables such as data stack pointer, heap pointer should be assigned to machine registers to obviate the need to load them into registers and save them in memory on entry and exit to every routine. Unfortunately, this is not possible in the programming system used.

4.1.4 Data Object Manipulation

There are three kinds of data object manipulation: accessing object, writing into object, and creating new object. For *FP*, heap operations outnumber stack operations. Accessing and creating operations are more important than writing into object, since heap objects are rarely modified.

FP objects are structured like trees, and when created, have to be flattened and fitted into the linear address space of the host machine. This makes necessary the use of a link field in a heap cell, creating additional demand for memory. While the use of a link field provides flexibility in allocating storage for sequences, locality of object storage is diminished. This is because adjacent elements of a sequence are not necessarily stored contiguously in memory. Loss of storage locality in a virtual memory machine incurs speed penalty in the form of page faults. For a machine with a memory cache, additional delay is caused by more cache misses.

4.2 Effect of Heap Size

The heap is a block of memory used to hold run-time objects. It is implemented as an array of cells. The size of the array is not dynamically variable. If the heap size is too small, then the system will not be able to run programs with a large data set. This is because the heap cannot hold all the objects needed simultaneously for computation to proceed. This can be so, even when the heap is garbage collected. There is simply not enough space to hold the object being created before existing objects can be discarded and garbage collected.

Given a set of programs and input data, there is a minimum heap size so that all programs can be run. Hence the size of the heap, beyond this minimum value, is a design parameter which has bearing on system performance.

Heap size can affect performance in two ways. Garbage collection, the way it is implemented using a marking algorithm, is directly influenced by heap size. Its cost, therefore, needs to be estimated. However, because of its invoked-on-demand nature, it is difficult to extrapolate and predict the cost of garbage collection from measured results. As is common for most measurements, measured results merely provide a glimpse of this cost in known instances of program execution.

Heap size also influences program execution by having an effect on run-time object manipulation. Object manipulation here takes the form of object creation and access. The cost of object creation depends on the cost of cell allocation which in turn, depends on the number of free cells in the heap. The cost of accessing objects, in a virtual memory system with a memory cache, depends on storage locality of objects accessed. Both costs are affected by the number of cells reclaimed each time garbage collection is done, since the ease of finding a free cell determines object creation cost and the abundance of free cells improves storage locality by storing related objects close together.

In this section, the effect of heap size is analyzed using a set of benchmark programs and input data set. The programs are given in appendix A, and consists of the following:

i. matrix multiplication, with an input of two twenty-by-twenty matrices,

ii. quick sort, sorting 500 random numbers,

iii. merge sort, also sorting 500 random numbers, and

iv. tower of hanoi, moving a stack of thirteen disks.

Using UNIX system call, *times()*, a simple timing facility is provided in the threaded interpreter to time top level function applications. Figures are reported for CPU time and garbage collection time. Three threaded interpreters are created, with a heap size of 4,000 cells, 8,000 cells, and 80,000 cells respectively. By using scripts, benchmarking processes are run on one UNIX host concurrently as background processes. This is done to ensure measurements for the three interpreters are done in the same environment, in the hope that the effect of heap size can be isolated from environmental ones.

Results of measurements are shown in Table 1 and 2. Each figure is an average of forty execution times. Table 1 shows the sum of CPU and garbage collection (GC) times for each function executed by each interpreter. Table 2 shows the CPU and GC components separately. It can be seen that GC is not as expensive as one might suspect, taking at most five percent of total execution time, in the cases shown.

From Table 1, the 8K interpreter performs consistently better than the 4K interpreter, but shows little difference, in terms of performance, from the 80K interpreter.

Looking at Table 2, where the CPU times are separated from the GC time, one would expect to find comparable CPU times for the three interpreters, and different GC times, with the smallest interpreter showing the most garbage collecting activity. This is partially true. GC times indeed decreases as heap size increases, but there are interesting variations in CPU times among the interpreters.

The 4K interpreter is slower because of the side effects caused by its small heap size, which requires more garbage collection during the course of computation. This not only raises GC time as measured but also CPU time. It can be speculated that the number of free cells reclaimed for a small heap is lower than that for a large heap. So free cells are harder to find and is reflected as higher CPU time. Another reason is that since garbage collection is more frequent, the system call, *times()*, used to collect execution statistics consumes resources which become significant.

Comparison of the 8K and 80K interpreters shows two conflicting results. The 8K interpreter is slower for programs, *matrix multiplication* and *quick sort*, but slightly faster for *merge sort* and *tower of hanoi*. Closer examination reveals that for the 80K interpreter, *matrix multiplication* and *quick sort* do not need garbage collection at all while *merge sort* and *tower of hanoi* both require GC. Thus, one might be tempted to argue that a large heap may cause performance degradation when GC is required. A possible reason why this may be so is offered in the next paragraph.

	Heap sizes		
	4K	8K	80K
matrix mult	17.80	15.78	14.28
quick sort	17.03	15.41	14.67
merge sort	75.00	70.45	71.46
tower	66.59	64.66	64.74
total	176.42	166.35	165.15

Table 1 Showing execution times (in seconds) of 4 programs for 3 heap sizes (in cells).

	Heap sizes		
	4K	8K	80K
matrix mult	16.21 + 1.59	15.05 + 0.73	14.28 + 0.0
quick sort	15.84 + 1.19	14.95 + 0.46	14.67 + 0.0
merge sort	71.37 + 3.63	67.93 + 2.52	69.99 + 1.47
tower	65.74 + 0.85	63.88 + 0.78	64.28 + 0.46
total	169.2 + 7.26	161.8 + 4.49	163.2 + 1.93

Table 2 Showing cpu + gc components (in seconds) of execution times in Table 1.

Note:
matrix mult takes two 20x20 matrices;
quick sort and merge sort work on 500 random numbers;
tower moves a stack of 13 disks

A big heap may lead to slower execution time if the effect of page faults is considered. Prior to GC, a heap provides free cell at the lowest cost, since free heap cells are allocated in the fastest way possible, consecutively without examining tag. A large heap has a greater capacity to do this than a smaller heap. After GC, free cells are scattered in the heap. Traversing the heap to find them causes frequent and regular page faults which slow down execution. This effect is more visible in a large heap than a small heap, since a big part of a small heap can be held in memory and not on paging device.

4.3 Comparison with Berkeley Interpreter

To provide some idea of the speed achieved by the threaded interpreter, the four benchmark programs are run and compared against *Berkeley FP* interpreter. It is known that such comparison is not fair, since *Berkeley FP* has the handicap of being based on *Franz*, the Lisp system. Comparison is useful, however, if the result is not used to extol the virtues of threaded interpretation, but merely to provide some idea of system responsiveness for a *FP* user who does not care about implementation detail.

The results of benchmark are shown in Table 3 and 4. *Berkeley FP* is able to generate Lisp code, which can be compiled, loaded, and run. Compiled code achieves a speed up of as much as fifteen times when compared to interpretive execution.

The threaded interpreter used has a heap of 1,000 cells and still costs less in garbage collection than either modes of *Berkeley FP*. Overall, the threaded interpreter is at least twice as fast as *Berkeley FP* compiled execution.

	a	b	c
matrix mult	10.9	6.9	2.43
quick sort	55.6	5.8	2.18
merge sort	198.4	15.2	5.75
tower	379.9	23.3	7.74
total	644.8	51.2	18.1

Table 3 Showing executio[illegible]mes for Berkeley FP and fpc. (see lege[illegible] below for details)

	a	b	c
matrix mult	7.4 + 3.5	4.2 + 2.7	2.19 + 0.24
quick sort	35.0 + 20.6	4.1 + 1.7	2.08 + 0.10
merge sort	121.3 + 77.1	12.1 + 3.1	5.55 + 0.20
tower	245.4 + 134.5	20.5 + 2.8	7.62 + 0.12
total	409.1 + 235.7	40.9 + 10.3	17.44 + 0.66

Table 4 Showing components of cpu + gc times for execution times in Table 3.

Note:
column a, Berkeley FP interpreting FP programs;
column b, Berkeley FP running compiled FP programs;
column c, threaded FP interpreter, heap size = 1000 cells

<u>matrix mult.</u> works on 2 10x10 matrices,
<u>quick sort</u> and <u>merge sort</u> takes a sequece of 100 numbers,
<u>tower</u> works on a stack Of 10 disks

4.4 Possible Improvements

While the threaded interpreter is faster, *Berkeley FP* offers more facilities for debugging and gathering statistics. These are easy to add on to the threaded interpreter.

A function tracing facility has been implemented by changing the code pointer in the lexicon entry of a traced function to point to a routine, which prints the object on top of the data stack at both function entry and exit times. By using the same technique, a facility to gather function execution statistics can also be implemented.

REFERENCES

[1] Backus, J.,"Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Comm. ACM, Vol.21, No.8, pp613-641, August 1978.

[2] Baden, S.,"Berkeley FP User's Manual, Rev. 4.1", January 1984.

[3] Worley, J., "The UCLA T-FP User Manual", June 1984.

[4] Lahti, D., "Applications of Functional Language", Master Thesis, 1981, UCLA.

[5] Lu, S.L., "A Compiler For A Functional Programming System", Master Thesis, November 1984, UCLA.

[6] Loeliger, R.G. "Threaded Interpretive Languages", Byte Books, 1981

[7] UNIX manual page for MALLOC(2).

[8] Vax-11 Architecture Reference Manual, Digital Equipment Corp., 1982

APPENDIX A

BENCHMARK PROGRAMS

```
# matrix multiplication, a matrix is represented by
# a sequence with elements as rows.
# input: <<a1,a2,,,aM>,<b1,b2,,,bN>>
#       where a's are of length N.

{matrixMult
        & MapInnerProd
# < <<a1,b'1>,<a1,b'2>,,,<a1,b'P>>
#       ,,,<<aM,b'1>,<aM,b'2>,,,<aM,b'P>> >
        @ & distl
# <<a1,B'>,<a2,B'>,,,<aM,B'>>
        @ distr @ [1, trans @ 2 ] }

{MapInnerProd
        & InnerProd }

{InnerProd
        | +
        @ & *
        @ trans }

#
# quick sort
#
{qSort (isShort -> id ;
                concat @ & qSort @ partition) }

{isShort <= @ [length, %1] }

{partition (isShort -> id ;
                [listSmall, tlr @ 1, listLarge] @ distl @ [1, tl]) }

{listSmall concat @ & pickSmall}

{pickSmall (>= -> [2] ; []) }

{listLarge concat @ & pickLarge}

{pickLarge (< -> [2] ; []) }

#
```

```
# merge sort
#
{mergeSort | merge}

{merge atEnd @ mergeHelp @ [ [], fixLists ]}

{fixLists &( atom -> [id] ; id ) }

{mergeHelp ( while and @ &( not @ null ) @ 2
                     ( firstIsSmaller -> takeFirst ;
                                         takeSecond )) }

{firstIsSmaller < @ [ 1@1@2, 1@2@2 ]}

{takeFirst [apndr @ [ 1, 1@1@2 ], [ tl@1@2, 2@2 ]] }

{takeSecond [apndr@[ 1, 1@2@2 ], [1@2, tl@2@2]] }

{atEnd ( firstIsNull -> concat @ [1,2@2] ;
                     concat @ [1,1@2])}

{firstIsNull null@1@2}


#
# tower of hanoi
#
{tower moveDisks @ [iota, [], []]}

{moveDisks
       (while

# done when 2 pegs are empty
              = @ [%2, countNull]

# move smallest disk, followed by the only legal move
              moveSmallest @ onlyLegalMove
       )
       @ moveSmallest }

{countNull ! + @ & testNull}

{testNull (null -> %1 ; %0)}

{moveSmallest
# smallest disk is on first peg
              (onFirst ->

# move it to second peg
                     [tl@1, apndl @ [%1,2], 3] ;

# smallest disk is on 2nd peg
```

```
                (onSecond ->

# move it to third peg
                        [1, tl@2, apndl @ [%1,3]] ;

# must be on third peg, move it to first peg
                        [apndl @ [%1,1], 2, tl @ 3]
                ))}

{onFirst (null -> %F ;
                = @ [%1, id]) @ first @ 1}

{onSecond (null -> %F ;
                = @ [%1, id]) @ first @ 2}

{onlyLegalMove          (onFirst ->
# smallest disk on first peg, do legal move on pegs 2 and 3
                        apndl @ [1, legalMove @ [2,3]] ;

# smallest disk on second peg, do legal move on pegs 1 and 3
                (onSecond ->
                        [1@2, 1, 2@2] @ [2, legalMove @ [1,3]] ;

# smallest disk on third peg, do legal move on pegs 1 and 2
                        [1@2, 2@2, 1] @ [3, legalMove @ [1,2]]
                ))}

{legalMove      (moveLeftP ->
                        [apndl @ [first @ 2, 1], tl @ 2] ;
                (moveRightP ->
                        [tl @ 1, apndl @ [first @ 1, 2]] ;
                        id
                ))}

{moveLeftP      > @ [getFirst @ 1, getFirst @ 2] }

{moveRightP     < @ [getFirst @ 1, getFirst @ 2] }

{getFirst       (null ->
                        %10000 ;
                        id
                ) @ first }
```

APPENDIX B

fpc User's Guide

fpc is an interactive threaded interpreter/compiler for the functional programming language, *FP*. It is smaller in size than Lisp-based interpreters and has been benchmarked to run at twice the speed of *Berkeley FP* interpreter executing compiled Lisp code.

This user's guide is divided into four sections. The first section deals with the syntax of acceptable *FP* programs. The second describes system commands implemented for file access, measurement, and debugging. The third section outlines implementation limitations and suggests a way of overcoming them if they become a problem. The last section mentions known bugs and discusses implementation issues which have not been resolved satisfactorily.

Syntax

This interpreter accepts the full set of *FP* constructs as described in *Berkeley FP User's Manual*. Users are referred to it for a full account. Differences from the Berkeley interpreter are described below.

1. *fpc* distinguishes between upper and lower case characters in its treatment of identifiers. Identifiers may consist of alphanumeric, '_', or '.' characters, but must begin with an alpha or '_' character. Identifiers containing other characters must be enclosed in double quotes.

2. *FP* primitives are accepted in lower case characters only.

3. Sequences can be represented using parentheses, '(' and ')', as well as angle brackets, '<' and '>'. Elements in a sequence may be separated by a white space or a comma.

4. Null sequence, "()" or "<>", must not have any white space character between enclosing parentheses or brackets.

System Commands

A subset of system commands supported by Berkeley FP is implemented. These are

1. *)fns,* to list function names for all user defined functions and their compiled code lengths in bytes, in the order they are found in the lexicon.

2. *)load <file>*, to cause the interpreter to accept input from the named file. Only one file may be specified, and may contain system commands as well as function definitions. The command, *)load,* itself can be nested up to four levels. File names containing path specification must be enclosed in double quotes. Wild card characters are not allowed.

3. *)timer on/off,* to enable/disable a simple timing facility. If enabled, the system will report CPU and garbage collection times at the end of top level function application. Format of times reported is in user and system components as obtained by the UNIX system call, *times()*. User time is the time spent executing user program in user address space; system time is time spent in kernel address space.

4. *)trace on/off <function>*, to enable/disable function tracing. If enabled, the system will, on entry to and exit from a traced function, display the name of the function, the dynamic nesting level of this call, and the data object on top of the data stack. Primitives are not traceable.

No command to save defined functions in a file is implemented. This is mainly because *fpc* does not keep source text of functions in the system. Saving compiled code is useless where defined functions need to be modified. Users therefore, should not develop functions by typing definitions directly at the interpreter. A text editor should be used to create a text file to hold function definitions. By using the job control mechanism of *C shell*, control can be transferred easily between the editor and *fpc*.

Implementation Notes

Various data structures in the interpreter are implemented as fixed-sized arrays. *fpc* does not handle array overflow gracefully. If an overflow occurs, an error message identifying the array is printed, the user is then returned to the shell. (Another reason why function definitions should always be kept in a file.) To overcome this problem, the offending array can be made larger and the system recompiled. Array overflow is usually caused by legitimate exhaustion of memory resource. However, bugs in function definition have been known to cause the data stack to overflow.

The data structures likely to overflow are listed below.

1. The data heap is implemented by a cell array. The size of the array is declared by the constant MAXCELL in the source file "heap.c".

2. The data stack is implemented with an array size as defined by the constant

MAXDS, which may be found in the file "eval.c".

3. The lexicon is an array known as *udf,* with MAXUDF slots, which is defined in the file "compile.c".

4. The byte array used to hold compiled code is declared in the file "codeBlk.c"; its size is MAXBYTES bytes.

Other data structures less likely to overflow are:

1. A stack holding head and tail pointers for a sequence being built. (The depth of the stack depends on how deep the sequence is nested.) The relevant declaration is in the file "heap.c"; the constant defining array size is MAXNEST.

2. A node array of MAXNODE nodes to hold a parse tree. This is defined in the file "heap.c".

3. A character array of MAXCHARS bytes to hold identifiers. The relevant file is "string.c".

Known Bugs and Rough Edges

These are briefly described below.

1. The functional form *CONSTANT* is not true *bottom preserving* For example, %(1 ? 3):10 returns the sequence (1 ? 3) instead of ?. Other primitives do preserve *bottom.*

2. The handling of *bottom* in the run-time routine, *appTOS,* called to build sequence is awkward. Typically, the routine is called to append the object on top of the data stack to a sequence being built. If all goes well, the calling routine returns the sequence as the result. However, if *appTOS* discovers it is ap-

pending *bottom* to a sequence, it will attempt to abort its caller and return *bottom* on the caller's behalf. It does this by writing *bottom* to the top of the data stack. It will then abort its caller by returning to the calling routine of its caller by popping a return address from the host's machine stack, and then executing a *return from subroutine* instruction. This scheme is designed to work in code compiled from *FP* functions. Returning to grandparent will not be successful, if a frame is on the machine stack instead of a return address, when the *return from subroutine* instruction is executed.

3. As each function is compiled, code is written to a scratch pad area before being moved to an allocated code block. There is no check to ensure that scratch pad does not overflow. Checking is not done since different routines are called to write code to the scratch pad for different *FP* constructs. There is no convenient place where checking can be centralized. Hence the size of the scratch pad has been made large, so that overflow is not likely.