Earlier version appears in: *Proceedings of the Fifth International Lisp Users and Vendors Conference, Boston, MA, 1995.*

Literate Engines in Lisp

Raja Sooriamurthi
Computer Science Department, Lindley Hall, 215
Indiana University, Bloomington, IN 47405
Email: `raja@cs.indiana.edu` *Phone:* 812-855-8702

Civilization advances by extending the number of important operations which we can perform without thinking about them.

– ALFRED NORTH WHITEHEAD (An introduction to Mathematics, 1911)

Abstract

An engine is a programming language abstraction that implements timed preemption. Engines form a useful mechanism where bounded computation is needed. We describe an implementation of engines in Common Lisp and illustrate its functionality. We conclude with a discussion on how engines are being used in an ongoing project about goal-driven explanation. This paper is also an exercise in writing a literate program in Lisp.

1 Introduction

The need for time bounded computation arises in several areas e.g., process scheduling in operating systems, heuristic search in AI, simulation. Primarily the ability to preempt a process from running and to be able to resume that process at a later time have been capabilities relegated to an operating system. In 1984 Haynes and Friedman [7, 8] introduced a programming language abstraction termed *engines* [1] for timed preemption. Engines allow a process to be preempted after a prespecified amount of computation has occurred. They also provide the ability to resume a suspended computation on demand. Engines were first introduced as an extension to the Scheme-84 interpreter [5]. Since then they have also been available in other Scheme systems (e.g., Chez scheme [3] and PC-scheme [2]). The Scheme implementations of engines are built upon the fact that in Scheme continuations are first-class objects and the programmer has access to the current continuation at any point by means of the reification procedure `call-with-current-continuation` (also called `call/cc`). Common Lisp does not directly support first class continuations. Hence a Scheme style implementation of engines in terms of continuations is not directly possible. However we could convert the Lisp code to Continuation Passing Style (CPS) [6] thereby explicitly creating and manipulating continuations. The engine implementation could then parallel that of Dybvig & Hieb [4].

In this paper we show how engines can be implemented in Common Lisp on top of a multitasking facility. Though multitasking is not part of ANSI Common Lisp most Lisp systems provide some version. To make things concrete we show our implementation of engines on top of the multitasking facility in Lucid Common Lisp [13]. We first discuss the functionality of engines and what is required to implement them. This is followed by a description of our implementation of engines in Lucid Common Lisp. This implementation can easily be translated to any other Lisp system supporting multitasking. Finally we discuss our current application of engines in a project on goal-driven explanation.

It should be noted that, conceptually engines are a more primitive facility than multi-tasking. In fact, in Scheme systems multi-tasking facilities are built on top of engines. The approach we took to implement engines described in this paper is due to (a) the lack of support for first class continuations and (b) the prevalence of multi-tasking systems in most versions of Common Lisp.

[1] Eugene Kohlbecker is credited with coining the term *engines* for this abstraction.

© 1995 Raja Sooriamurthi

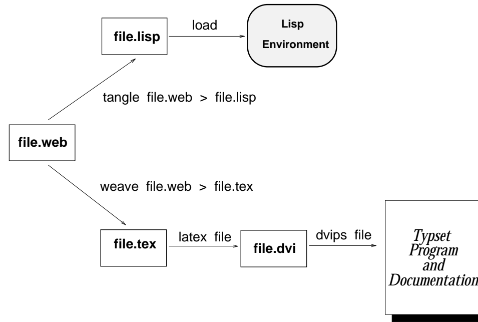

Figure 1: Literate Programming = Structured Code + Structured Documentation

2 Literate Programming in Lisp

The programs in this paper have been written in a literary style using Norman Ramsey's `noweb` [14]. Literate programming is a methodology introduced by Knuth in trying to attain his goal that programs should be *works of literature* and fun to read [10]. Figure-1 outlines the process of literate programming.

A literate programmer combines both the documentation and code of a program into a single unit known as a *web*. Knuth's TEX and METAFONT are perhaps the best (and largest) known examples of this style of programming. From a web through a process known as *tangling* the program code can be extracted. Through *weaving* typeset documentation can be extracted. This paper and the Lisp code for our engine system have both been derived from the same `noweb` file. The interactive nature of Lisp combined with the support of the `noweb-mode` Emacs mode by Thorsten Ohl makes Lisp programming under Emacs an effective literary programming environment [16].

The outline of the code in this paper is as follows:

2a ⟨*Literate Engines in Lisp* 2a⟩≡
  ⟨*A simple prime number generator* 2b⟩
  ⟨*A "primed" engine example* 3⟩
  ⟨*Make engine* 5d⟩
  ⟨*First to complete* 7a⟩
  ⟨*Example of utility bounded search* 8b⟩

3 Functionality of Engines

An engine is created by means of the procedure `make-engine`. This takes a *thunk* as its argument which represents the computation that is to be performed. For instance consider the following procedure which computes the first n prime numbers:

2b ⟨*A simple prime number generator* 2b⟩≡ (2a)

```
(defun prime-generator (n &aux i (j 0))
  (loop for i from 2
        when (= j n)
        do (return ls)
        when (prime-p i)
        do (incf j)
        and do (format t "Prime (~a) is ~a~%" j i)
        and collect i into ls))

;;; a simple test for primality

(defun prime-p (n)
```

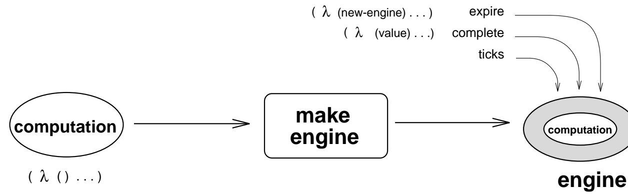

Figure 2: A conceptual view of engines.

```
(or (= n 2)
    (loop for i from 2 to (1+ (isqrt n))
          when (zerop (rem n i))
          do (return nil)
          finally (return t))))
```

Defines:
`prime-generator`, used in chunk 3.

We can transform the computation to calculate the first 15 primes into an engine by means of:

```
(make-engine #'(lambda () (prime-generator 15)))
```

This returns an engine for the computation `(prime-generator 10)`. Figure-2 diagrams the creation and usage of engines.

An engine is implemented as a function of three arguments: `(<engine> <ticks> <complete> <expire>)`

Ticks	A non-negative integer denoting the amount of computation to be performed by the engine.
Complete	A procedure of one argument specifying what to do if the computation completes [2].
Expire	A procedure of one argument specifying what to do if the ticks are consumed before the computation completes. Its argument is a new engine capable of continuing the computation from the point of suspension.

Metaphorically the `ticks` represent the amount of "fuel" given to an engine. A tick is associated with some amount of *computation* measured, for example, in terms of an internal clock, the number of instructions executed, the number of procedure calls made etc. A tick need not be associated with the same amount of computation each time. All that is required is that more ticks be associated with more computation.

When an engine is run if its computation completes within its specified amount of ticks then the answer is returned via the `complete` procedure. If the computation does not complete within the specified number of ticks then a new engine is created and passed on to the `expire` routine. This engine can then be invoked at a later time to resume the computation.

Following is an example using the prime number generator. We first define the computation to be performed, the completion and expiration routines.

3 ⟨*A "primed" engine example* 3⟩≡ (2a)

```
;;; The computation to be performed
(defun computation () (prime-generator 15))

;;; The procedure that manipulates the final answer
(defun complete (v)
  (format t "Value is = ~a~%" v)
  v)
```

[2] This is different from Scheme engines wherein the `complete` procedure takes two arguments, the number of ticks remaining unconsumed when the computation completes and the result of the computation.

```
> (let ((eng (make-engine #'computation)))
     (funcall eng 10 #'complete #'expire))
Prime (1) is 2
Prime (2) is 3
Prime (3) is 5
** Making new engine **
Expired
#<Compiled-Function (:internal make-engine eng) BAF506>

> (funcall *global-eng* 10 #'complete #'expire)
Prime (4) is 7
Prime (5) is 11
Prime (6) is 13
Prime (7) is 17            ;; The engine is called again
Prime (8) is 19
Prime (9) is 23
** Making new engine **
Expired
#<Compiled-Function (:internal make-engine eng) BAF506>
> (funcall *global-eng* 10 #'complete #'expire)
Prime (10) is 29
Prime (11) is 31
Prime (12) is 37           ;; The engine is called one last time
Prime (13) is 41
Prime (14) is 43
Prime (15) is 47
 ;; The returned answer
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
```

Figure 3: Trace of the execution of the "primed" engine.

```
;;; The routine that handles the intermediate engines.
;;; It assigns the intermediate engines to the global variable *global-eng*
(defun expire (new-eng)
  (format t "Expired~%")
  (setf *global-eng* new-eng))
```

Defines:
  `complete`, used in chunk 6.
  `expire`, used in chunk 6.
Uses `prime-generator` 2b.

Given these definitions Figure-3 gives a trace of the behavior.

4 An Implementation of Engines in Lisp

To implement engines we need two capabilities (1) the ability to stop a computation midway and save its state and (2) the ability to resume a suspended computation. Lucid Common Lisp (and other Common Lisps) provides a multitasking facility [13]. Using this facility a group of processes can be created. Each of the processes are then run in a prioritized round robin fashion for a specified amount of time. A process that is either waiting to run or is running is termed an active process. An active process may be deactivated upon which it will not be considered for running until activated again. A multi-tasking facility thus provides the requirements to implement engines. The requisite code is discussed below:

Given the engine computation in the form of a thunk we create a process that will run it.

5a ⟨*Make process* 5a⟩≡ (5d)
```
(make-process
 :name (symbol-name (gensym "ENG-"))
 :stack-size 2000
 :function ⟨Engine computation 5b⟩
 :wait-function ⟨Wait function 5c⟩)
```

If the engine computation does complete then its result is stored in the processes property list. So the actual bounded computation performed by the engine is the invocation of the thunk followed by a property list access.

5b ⟨*Engine computation* 5b⟩≡ (5a)
```
#'(lambda ()
    (with-scheduling-allowed
     (setf (getf (process-plist proc) 'answer)
           (funcall thunk))))
```

To suspend the process the moment it is created we use a wait function which will initially return false.

5c ⟨*Wait function* 5c⟩≡ (5a)
```
#'(lambda () go))
```

We now have the definition of `make-engine` as:

5d ⟨*Make engine* 5d⟩≡ (2a)
```
(defun make-engine (thunk)
  (let ((go nil))   ;; used to initially wait the process
    (let (proc)
      ;; create the engine
      (setf proc ⟨Make process 5a⟩)
      ;; turn off the process
      (deactivate-process proc)
      ;; The engine per se
      ⟨Create the engine 6a⟩)))
```

By deactivating the process we remove it from the list that is being actively scanned by the scheduler. (A deactivated process is a hibernating process.)

Before we can create the engine we need a mechanism to perform the timed preemption. This is done by means of another process. The timer process is waited with the same function as the engine. It is started after the engine process and when it runs it deactivates the engine process. (If the host Common Lisp were to support an explicit lightweight timer facility a timer process would not be needed.)

5e ⟨*Timer process* 5e⟩≡ (6a)
```
(make-process
 :name "timer-proc"
 :function
 #'(lambda ()
     (unless (not (process-alive-p proc))
       (deactivate-process proc)))
 :wait-function #'(lambda () go))
```

Finally the code that creates the engine itself.

6a ⟨*Create the engine* 6a⟩≡ (5d)

```
(labels ((eng (ticks complete expire)
              ;; set the timer to run for ticks units
              (let-globally ((*scheduling-quantum* ticks))
                ;; Activate the process
                (activate-process proc)
                ;; create the timer
                ⟨Timer process 5e⟩
                ;; turn on the wait functions
                (setf go t)
                ;; the process can either complete or
                ;; it will have been preempted
                (process-wait "For engine to run"
                              #'(lambda () (not (process-active-p proc))))
                ⟨Complete or expire 6b⟩)))
  ;; return the engine
  #'eng)
```

Uses `complete` 3 and `expire` 3.

When the engine has been made inactive it has either completed its computation or it has been suspended. The occurrence of the property **answer** in the engines property list indicates completion.

6b ⟨*Complete or expire* 6b⟩≡ (6a)

```
(cond ((member 'answer (process-plist proc))
       ;; then we are done
       (funcall complete
                (getf (process-plist proc) 'answer)))
      ;; else we create and return a new engine
      ;; aha! just return the same proc again!
      (t (format t "~&** Making new engine **~%")
         (funcall expire #'eng)))
```

Uses `complete` 3 and `expire` 3.

This completes a simple definition of engines in terms of multi-tasking processes.

5 Using Engines for Goal-Driven Explanation

The ability of engines to perform bounded computation have been used to implement multitasking OS kernels in Scheme systems. In this section we describe the proposed use of engines in an ongoing project for goal-driven explanation, GOBIE[3]. (Further details about GOBIE and the motivations behind the project may be found in [17, 12].)

GOBIE is a system for performing explanation in a dynamic goal-driven manner. It is an object-oriented system written in CLOS [1, 9]. The system consists of a planner functioning in a simple simulated world and a case-based explainer [15] integrated across a blackboard. Case-based explanation applies the problem solving paradigm of case-based reasoning [11] to the problem of formulating explanations: new explanations are formed by adapting explanations that had been applied to similar prior problems. The schematic representation of the system is given in Figure-4.

In GOBIE the explanation process is shaped by ongoing and strategic decisions as the explainer interacts with its environment (the simulated world). Our model of explanation, termed *goal-driven interactive explanation* (GDIE) [12] allows incremental information obtained by interaction with the environment to influence the explainer's goals and dynamically re-focus the explanation process. Given a goal the planner comes up with a collection of plans that may achieve the goal. GOBIE is a utility-based system. Hence the

[3] GOal Based Interactive Explanation

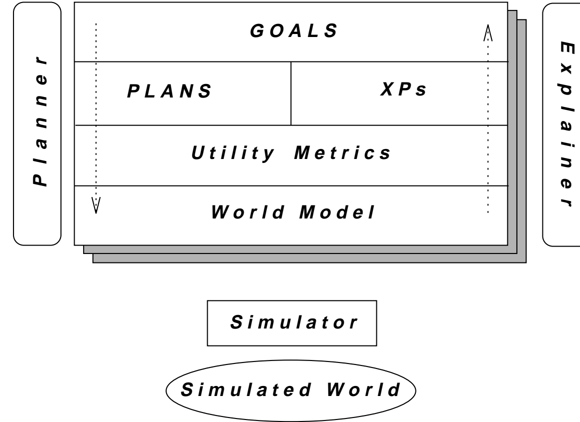

Figure 4: The blackboard framework of GOBIE. The system consists of a planner functioning in a simulated world and a case-based explainer integrated across a blackboard.

plan that is deemed to be cheaper is selected and executed in a simulated world. If the execution of a plan step fails the explainer tries to diagnose the fault and helps the planner recover from the failure. But if the fault is deemed irrecoverable then the planner abandons its current plan and attempts an alternative plan.

For example, consider the following implemented scenario. The planner is given the goal of catching a flight. To do this the planner has to get to the air-port. The planner has two alternative plans of either driving or taking a taxi. Choosing the cheaper alternative of driving it simulates the steps involved in that plan. Events in the simulator are probabilistically driven. In this scenario the simulator is set up such that the car fails to start. The case-based explainer then tries to diagnose the fault based on prior cases it has seen. It recalls three prior problems: a problem with the spark-plugs, a dead-battery and fouled fuel pipes. Each one of these candidate explanations are to be examined in a utility based goal driven manner. In general in an explanation task we have a collection of alternative hypothesis that have to be examined. In GOBIE these alternatives are examined with *varying amounts of effort* depending on the motivation of the system and the specific costs and circumstantial situation of the explanation process. We propose to extend the current utility based examination of candidates with engines. Each alternative will be examined by an engine. The amount of "fuel" given to each engine will be representative of how much effort the system wants to expend in examining a particular candidate.

Suppose we have alternatives a_1, a_2, ... a_n. Then the type of search we are interested in could be expressed as: `(utility-bounded-search` a_1 a_2 a_3 ... a_n`)`. An engine is created for each alternative and the engines are run in parallel with varying amounts of effort.

`utility-bounded-search` macro-expands to

```
(first-to-complete #'(lambda () a1) #'(lambda () a2)  . . .  #'(lambda () an))
```

Which in turn is implemented as (the code uses some auxiliary queue functions which are not shown.):

7a ⟨*First to complete* 7a⟩≡ (2a)

```
(defun first-to-complete (&rest proc-list)
  (let ((engines (queue)))
    (labels ((run () ⟨Run each engine 8a⟩))
      ⟨Create a group of engines 7b⟩
      (run))))
```

For each alternative that we have to examine we create an engine and store them in a queue.

7b ⟨*Create a group of engines* 7b⟩≡ (7a)

```
(loop for proc in proc-list
      do (enqueue (make-engine proc) engines))
```

The engines in the queue are run in sequence. The exact sequence in which the various engines are run (and hence the various alternatives explored) could be dynamically determined.

8a ⟨*Run each engine* 8a⟩≡ (7a)
```
(and (not (empty-queue-p engines))
     (funcall (dequeue engines)
              10
              #'(lambda (v) (or v (run)))
              #'(lambda (e) (enqueue e engines)
                   (run))))
```

As an example, suppose `(fact n)` computes $n!$. Then:

8b ⟨*Example of utility bounded search* 8b⟩≡ (2a)
```
(utility-bounded-search (fact 50) (fact1 7) (fact 6) (fact 10)))
```

evaluates to that expression which completes its computation first. Typically this will be be `(fact 6)` but since the amount of computation associated with a tick isn't fixed, at times, it can also be one of the other alternatives.

6 Conclusions

Engines are a convenient abstraction for implementing bounded computation. In the spirit of Alfred Whitehead's quote at the beginning of this paper they are a versatile abstraction building abstraction. In Scheme engines are normally implemented in terms of continuations. The absence of first class continuations in Common Lisp requires alternative implementations for engines. We have presented an implementation of engines on top of a multi-tasking facility. We have also described how such engines can help implement bounded computation in a goal-driven explanation system. In future work we plan to implement utility-directed processing using engines in other components of our goal-driven explanation system GOBIE. As part of this work we also experienced that the concept of literate programming meshes in quite well with the dynamic nature of Lisp prototyping.

Acknowledgments: My thanks to Prof. David Leake with whom I have been working on goal-driven explanation and the GOBIE system. Thanks also to the anonymous reviewers for their helpful comments.

Chunks:

⟨*A "primed" engine example* 3⟩
⟨*A simple prime number generator* 2b⟩
⟨*Complete or expire* 6b⟩
⟨*Create a group of engines* 7b⟩
⟨*Create the engine* 6a⟩
⟨*Engine computation* 5b⟩
⟨*Example of utility bounded search* 8b⟩
⟨*First to complete* 7a⟩
⟨*Literate Engines in Lisp* 2a⟩
⟨*Make engine* 5d⟩
⟨*Make process* 5a⟩
⟨*Run each engine* 8a⟩
⟨*Timer process* 5e⟩
⟨*Wait function* 5c⟩

Index:

`complete`: 3, 6a, 6b
`computation`: 3
`expire`: 3, 6a, 6b
`prime-generator`: 2b, 3
`prime-p`: 2b

References

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. Technical Report Document 88-002R, X3J13, 1988. Also appears in *Lisp and Symbolic Computation* **1** 3/4, January, 1989 245–394 and as Chapter 28 of [18], 770–864.

[2] Computer Science Laboratory, Texas Instruments Inc., Dallas, Texas. *TI Scheme language reference manual*, 1986. (TI Scheme was also released as PC-Scheme.).

[3] R. Kent Dybvig. *The Scheme Programming Language*. Prentice Hall, 1987. (Second edition in preparation. Expected in October 1995.).

[4] R. Kent Dybvig and Hieb Robert. Engines from continuations. Technical Report 254, Indiana University, Computer Science Department, July 1988.

[5] Daniel P Friedman, Christopher T. Haynes, Eugene Kohlbecker, and Mitchell Wand. The scheme 84 interim reference manual. Technical Report 153, Indiana University, Computer Science Department, June 1985.

[6] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press / McGraw Hill Press, 1992.

[7] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 18–24, 1984.

[8] Christopher T Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Journal of Computer Languages*, 12(2):109–121, 1987.

[9] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989.

[10] Donald E. Knuth. *Literate Programming*. CSLI, Stanford University, Stanford, CA, 1992.

[11] Janet Kolodner. *Case-Based reasoning*. Morgan Kaufmann, 1994.

[12] David B. Leake. Issues in goal-driven explanation. In Mariie desJardins and Ashwin Ram, editors, *Working notes of the AAAI Spring Symposium in goal-driven learning*, pages 72–79, 1994.

[13] Lucid Inc. *Lucid Common Lisp: Advanced User's Guide*, 2nd edition, September 1991.

[14] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, Sept 1994.

[15] Roger C. Schank, Alex Kass, and Christopher K. Riesbeck. *Inside Case-Based Explanation*. Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1994.

[16] Raja Sooriamurthi. A scheme word count program — a tutorial introduction to literate programming in scheme. Unpublished tutorial handout, September 1995.

[17] Raja Sooriamurthi and David B. Leake. An architecture for goal-driven explanation. In John H. Stewman, editor, *Proceedings of the eigth Florida Artificial Intelligence Research Symposium*, pages 218–222. Florida Artificial Intelligence Research Symposium, April 1995.

[18] Guy L. Steele Jr. *Common Lisp the language*. Digital Press, 2nd edition, 1990. Also available online from the CMU AI repository at `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html`.