Design and Implementation of Programming Languages

Prof. Mehran S. Fallah

Assignment #2

Induction / Semantic Styles / Evaluation / Lambda Calculus / Denotational Semantics / Functional & Imperative Languages

Done with **love** by

Ali Gholami

with Stu. #

9531504

Department of Computer Engineering and Information Technology

Amirkabir University of Technology

November 2017

The following are problems from the book, **Types and programming languages** by **Benjamin Pierce.**

3.5.10 Definition: The multi-step evaluation relation $\rightarrow *$ is the reflexive, transitive closure of one-step evaluation. That is, it is the smallest relation such that (1) if $t \rightarrow t'$ then $t \rightarrow * t'$ (2) $t \rightarrow * t$ for all t, and (3) if $t \rightarrow * t'$ and $t' \rightarrow * t''$, then $t \rightarrow * t''$. Rephrase the above definition as a set of inference rules.

Answer.

First of all, let's describe the closure of a relation. The closure of a relation is the smallest relation that has the attributes of that relation. Now deriving the set of inference rules for the above multi-step evaluation relation:

$$\frac{t \rightarrow t'}{t \rightarrow * t'} \qquad \frac{}{t \rightarrow * t} \qquad \frac{t \rightarrow * t' \quad t' \rightarrow * t''}{t \rightarrow * t''}$$

3.5.17 Two styles of operational semantics are in common use. The one used in this book is called the small-step style, because the definition of the evaluation relation shows how individual steps of computation are used to rewrite a term, bit by bit, until it eventually becomes a value. On top of this, we define a multi-step evaluation relation that allows us to talk about terms evaluating (in many steps) to values. An alternative style, called big-step semantics (or sometimes natural semantics), directly formulates the notion of "this term evaluates to that final value," written t ⇓ v. The big-step evaluation rules for our language of Boolean and arithmetic expressions look like this:

$$v \Downarrow v$$

$$\frac{t_1 \Downarrow true \quad t_2 \Downarrow v_2}{if\ t_1\ then\ t_2\ else\ t_3 \Downarrow v_2}$$

$$\frac{t_1 \Downarrow false \quad t_3 \Downarrow v_3}{if\ t_1\ then\ t_2\ else\ t_3 \Downarrow v_3}$$

$$\frac{t_1 \Downarrow nv_1}{succ\ t_1 \Downarrow succ\ nv_1}$$

$$\frac{t_1 \Downarrow 0}{pred\ t_1 \Downarrow 0}$$

$$\frac{t_1 \Downarrow succ\ nv_1}{pred\ t_1 \Downarrow nv_1}$$

$$\frac{t_1 \Downarrow 0}{iszero\ t_1 \Downarrow true}$$

$$\frac{t_1 \Downarrow succ\ nv_1}{iszero\ t_1 \Downarrow false}$$

Show that the small-step and big-step semantics for this language coincide, i.e. $t \rightarrow * \ v$ iff $t \Downarrow v$.

Answer.

The two directions of the above integration of semantics need to be solved. For the first direction I'll use the first lemma given below.

Lemma 1.

If $t_1 \rightarrow * \ t'_1$then $if\ t_1\ then\ t_2\ else\ t_3 \rightarrow * \ if\ t'_1\ then\ t_2\ else\ t_3$ and similarly for the other term constructors.

Proof

Easy induction by replacing v_2 with t_2.

In this section I'll prove the other direction which is:

$$if\ t \Downarrow v\ then\ t \rightarrow * \ v$$

We're going to implement a case analysis for the above proposition. By induction on the derivation of $t \Downarrow v$, with a case analysis on the final rule used.

Case B-Value:

$$t = v$$

Immediate.

Case B-IfTrue:

$$\frac{t = if\ t_1\ then\ t_2\ else\ t_3 \quad t_1 \Downarrow true \quad t_2 \Downarrow v}{}$$

The result will be v according to the lemma 1

3.5.18 Suppose we want to change the evaluation strategy of our language so that the then and else branches of an if expression are evaluated (in that order) before the guard is evaluated. Show how the evaluation rules need to change to achieve this effect.

Answer.

Before getting into the answer, I consider the room for defining the term "guard". Guards are simple terms whose result are Boolean. So in order to skip the guards, we need to evaluate the expressions when we parse if/else structure. To achieve this, we can have the following propositional induction strategy:

$$\frac{t_1 \Downarrow t_2 \quad t_2 \Downarrow v_2 \quad t_3 \Downarrow v_3}{if\ t_1 then\ t_2\ else\ t_3 \Downarrow t_2}$$

5.2.2 Find another way to define successor function on Church numerals.

Answer.

$$scc = \lambda n.\ \lambda s.\ \lambda z.\ n\ s\ (s\ z)$$

5.2.3 Similarly, addition of Church numerals can be performed by a term **plus** that takes two Church numerals, m and n, as arguments, and yields another Church numeral—i.e., a function—that accepts arguments s and z, applies s iterated n times to z (by passing s and z as arguments to n), and then applies s iterated m more times to the result:

$$plus = \lambda m.\lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z)$$

The implementation of multiplication uses another trick: since plus takes its arguments one at a time, applying it to just one argument n yields the function that adds n to whatever argument it is given. Passing this function as the first argument to m and c_0 as the second argument means "apply the function that adds n to its argument, iterated m times, to zero," i.e., "add together m copies of n."

$$times = \lambda m.\lambda n.\ m\ (plus\ n) c_0$$

Answer.

$$times2 = \lambda m.\ \lambda n.\ \lambda s.\ m\ (n\ s)$$

5.2.4 Define a term for raising one number to the power of another.

Answer.

$$power = \lambda m.\ \lambda n.\ m\ (times\ n)\ c_1)$$

5.2.5 This definition works by using m as a function to apply m copies of the function ss to the starting value zz. Each copy of ss takes a pair of numerals pair $c_i\ c_j$ as its

argument and yields pair $c_i\ c_{j+1}$ as its result (see Figure 51). So applying ss, m times, to pair $c_0\ c_0$ yields pair $c_0\ c_0$ when $m = 0$ and pair $c_m - 1$ cm when m is positive. In both cases, the predecessor of m is found in the first component.

$$zz = pair\ c_0 c_0$$

$$ss = \lambda p.\ pair\ (snd\ p)\ (plus\ c_1 (snd\ p))$$

$$prd = \lambda m.\ fst\ (m\ ss\ zz)$$

Use prd to define a subtraction function.

Answer.

Using the term prd, the subtraction can be achieved by applying prd b times to a.

$$subtraction = \lambda b.\ \lambda a.\ b\ prd\ a$$

5.2.7 Write a function equal that tests two numbers for equality and returns a Church Boolean. For example:

$$equal = c_3 c_3;$$

results in

$$\lambda t.\ \lambda f.\ t$$

and so on.

Answer.

If m is equal to n and n is also equal to m, they will be the same.

$$equal = \lambda m.\ \lambda n.\ and\ \big(iszero\ (m\ prd\ n)\big)\ \big(iszero\ (n\ prd\ m)\big)$$

5.2.8 A list can be represented in the lambda calculus by its $fold$ function. (OCaml's name for this function is $fold\_left$; it is also sometimes called reduce.) For example, the list $[x, y, z]$ becomes a function that takes two arguments c and n and returns $c\ x\ (c\ y\ (c\ z\ n))$. What would the representation of nil be? Write a function $cons$ that takes an element h and a list (that is, a fold function) t and returns a similar representation of the list formed by prepending h to t. Write $isnil$ and $head$ functions, each taking a $list$ parameter. Finally, write a $tail$ function for this representation of lists (this is quite a bit harder and requires a trick analogous to the one used to define prd for numbers).

Answer.

$$nil = pair\ tru\ tru$$

$$cons = \lambda h. \lambda t.\ pair\ fls\ (pair\ h\ t)$$

$$head = \lambda z.\ fst\ (snd\ z)$$

$$tail = \lambda z.\ snd\ (snd\ z)$$

$$isnil = fst$$

5.2.9 Define a function churchnat that converts a primitive natural number into the corresponding Church numeral.

Answer.

This can be achieved using a recursive function. We need to find the predecessor of the natural number until we reach zero, then we return a church zero and apply the successor function on the church zero. This will give us the corresponding church numeral for the given natural number.

$$churchnat = \lambda f.\lambda m.if\ iszero\ m\ then\ c_0\ else\ succ(\ f(prd\ m))$$

5.2.11 Use fix and the encoding of lists from Exercise 5.2.8 to write a function that sums lists of Church numerals.

Answer.

We ought to define a function $sumlist$ with the fixed-point of ff in order to find the addition of terms inside a list given to us. Therefore, we will start from the first element and iterate through the rest of the list recursively. Each time we'll grab the $head$ of the list, add it to our current $value$ and call the f for the rest of the lits.

$$ff = \lambda f.\ \lambda l.\ test(isnil\ l)(\lambda x.\ c_0)\Big(\lambda x.\ \big(plus\ (head\ l)\big(f\ (tail\ l)\big)\big)\Big)\ c_0;$$

$$sumlist = fix\ ff;$$

The following are problems from the book, **Concepts in programming languages** by **John C. Mitchel**

4.4 **Symbolic Evaluation**

The Algol-like program fragment,

```
function f(x)
    return x+4
end;
function g(y)
    return 3-y
end;
f(g(1));
```

can be written as the following lambda expression:

$$((\lambda f.\ \lambda g.\ f(g\ 1))\ (\lambda x.x\ +4))(\lambda y.3-y)$$

Reduce the expression in two different ways, as described below.

a) Reduce the expression by choosing at each step, the reduction that eliminates a λ as far to the $left$ as possible.
b) Reduce the expression by choosing at each step, the reduction that eliminates a λ as far to the $right$ as possible.

Answer. (a)

$$\Big((\lambda f.\lambda g.f(g\ 1))(\lambda x.x+4)\Big).(\lambda y.3-y)$$

$$\begin{gathered}\big(\lambda g.(\lambda x.x+4)(g\ 1)\big)(\lambda y.3-y)\\ ((\lambda x.x+4)(\lambda y.3-y\ 1))\\ (\lambda y.3-y\ 1)+4\\ (3-1)+4\end{gathered}$$

And the process will stuck at the above normal form.

Answer. (b)

$$\Big((\lambda f.\lambda g.f(g\ 1))(\lambda x.x+4)\Big).(\lambda y.3-y)$$

$$\begin{gathered}(\lambda g.(\lambda x.x+4)(g\ 1))(\lambda y.3-y)\\ (\lambda g.(g\ 1)+4)(\lambda y.3-y)\\ \big((\lambda y.3-y)1\big)+4\\ (3-1)+4\end{gathered}$$

And the process will stuck at the above normal form again.

4.5 **Lambda Reduction with Sugar**

Here is a "sugared" lambda expression that uses let declarations:

$$let\ compose = \ \lambda f.\ \lambda g.\ \lambda x.f(\ g\ x\ )\ in$$
$$let\ h = \ \lambda x.\ x + \ x\ in$$
$$compose\ h\ h\ 3$$

The "DE sugared" lambda expression, obtained when each let $z = U$ in V is replaced with

$$(\lambda z.V)U\ is$$
$$(\lambda compose.$$
$$(\lambda h.\ compose\ h\ h\ 3)\ \lambda x.x\ + x\ )$$
$$\lambda f.\ \lambda g.\ \lambda x.f(g\ x)$$

This is written with the same variable names as those of the let form to make it easier to read the expression.

Simplify the DE sugared lambda expression by using reduction. Write one or two sentences explaining why the simplified expression is the answer you expected.

Answer.

We simply apply the one-step evaluation rule to the above DE sugared lambda expression and we get the following:

$$(\ (\lambda h.\ \lambda f.\ \lambda g.\ \lambda x.\ f(g\ x)\ h\ h\ 3)\ \lambda x.x + x)$$
$$(\ \lambda f.\ \lambda g.\ \lambda x.\ f(g\ x)\ \lambda x.x + x\ \ \lambda x.x + x\ \ 3)$$

4.6 **Translation into Lambda Calculus**

A programmer is having difficulty debugging the following C program. In theory, on an "ideal" machine with infinite memory, this program would run forever. (In practice, this program crashes because it runs out of memory, as extra space is required every time a function call is made.)

```
int f(int (*g)(...)){ /* g points to a function that returns an int */
   return g(g);
}
int main(){
  int x;
  x = f(f);
  printf("Value of x = %d\n", x);
  return 0;
}
```

Explain the behavior of the program by translating the definition of f into lambda calculus and then reducing the application $f(f)$.This program assumes that the type checker does not check the types of arguments to functions.

Answer.

Simply speaking, we can write down the lambda calculus version like this:

$$f = \lambda g.gg$$

$$(f)f = (\lambda g.gg)(f)$$

$$(f)f = ff$$

The application above cannot be evaluated at all and the function f will stuck at some point.

4.8 **Denotational Semantics**

The text describes a denotational semantics for the simple imperative language given by the grammar

$$P ::= x := e \mid P_1; P_2 \mid \text{if } e \text{ then } P_1 \text{ else } P_2 \mid \text{while } e \text{ do } P.$$

Each program denotes a function from states to states, in which a state is a function from variables to values.

a) Calculate the meaning $C[\![x := 1; x := x + 1;]\!](s_0)$ in approximately the same detail as that of the examples given in the text, where $s_0 = \lambda v \in variables.\, 0$, giving every variable the value 0.
b) Denotational semantics is sometimes used to justify ways of reasoning about programs. Write a few sentences, referring to your calculation in part (a), explaining why $[\![x := 1; x := x + 1;]\!](s) = C[[x := 2;]](s)$ for every state s.

Answer. (a)

$$c[[x := 1; x := x + 1;]](s_0)$$
$$c[[x := x + 1]](x \to 1(s_0))$$
$$c[\![x := x]\!](s_1) + 1$$
$$S_2(x \to 1) + 1$$
$$1 + 1 = 2 = C[[x := 2]](s)$$

Answer. (b)

According to the denotational semantics here, the evaluation firstly will change current state to the state which x is equal to 1. After setting the x to 1, the evaluation continues to change the state which x is incremented by 1. So the result in the new state will make x have the value of 2. So in that case, the x will be equal to 2 for any state the program has.

4.9 **Semantics of Initialize-Before-Use**

A nonstandard denotational semantics describing initialize-before-use analysis is presented in the text.

a) What is the meaning of

$$C\big[[x := 0; y := 0; if\ x = y\ then\ z := 0\ else\ w := 1]\big](s_0)$$

In the state $s_0 = \lambda y \in variables.uninit?$ show how to calculate the your answer.

b) Calculate the meaning

$$C[[if\ x = y\ then\ z := y\ else\ z := w]](s)$$

In state s with $s(x) = init, s(y) = init, and\ s(v) = uninit$ or every other variable v.

Answer. (a)

Now writing the status of the first given command to find its meaning:

$$C\big[[x := 0; y := 0; if\ x = y\ then\ z := 0\ else\ w := 1]\big](s_0)$$
$$C[[y := 0; if\ x = y\ then\ z := 0\ else\ w := 1]](x \to 0(s_0))$$
$$C[[if\ x = y\ then\ z := 0\ else\ w := 1]](y \to 0(s_1))$$
$$if\ E\big[[x = y]\big](s_2) = error\ or\ C\big[[z := 0]\big](s_2) = error\ or\ C\big[[w := 1]\big] = error\ then\ error\ else\ C\big[[z := 0]\big](s_2) + C[[w := 1]](s_2)$$
$$if\ s_2(z) = s_2(w) = init\ then\ init\ else\ uninit\ (which\ will\ be\ init)$$

Answer. (b)

Now running the second command to find its meaning with respect to the given assumptions:

$$C[[if\ x = y\ then\ z := y\ else\ z := w]](s)$$
$$if\ E\big[[x = y]\big](s) = error\ or\ C\big[[z := y]\big] = error\ or\ C\big[[z := w]\big](s) = error\ then\ error\ else\ C\big[[z := 0]\big](s) + C[[z := w]](s)$$
$$error\ (because\ of\ z := w)$$

4.10 **Semantics of Type Checking**

This problem asks about a nonstandard semantics that characterizes a form of type analysis for expressions given by the following grammar:

$$e ::= 0|1|true|false|x|e + e|if\ e\ then\ e\ else\ e|\ let\ x{:}\ \tau = e\ in\ e$$

In the let expression, which is a form of local declaration, the type τ may be either int or $bool$. The meaning $V\big[[e]\big](\eta)$ of an expression e depends on an environment.

An environment η is a mapping from variables to values. In type analysis, we use three values,

$$Values = \{integer, boolean, type\_error\}$$

Intuitively, $V\big[[e]\big](\eta) = integer$ means that the value of expression e is an integer (in environment η) and $V\big[[e]\big](\eta) = type\_error$ means that evaluation of e may

involve a type error. Here are the semantic clauses for the first few expressions form.

$$V[\![0]\!]\eta = integer$$
$$V[\![1]\!]\eta = integer$$
$$V[\![true]\!]\eta = boolean$$
$$V[\![false]\!]\eta = boolean$$

a) Show how to calculate the meaning of the expression $if\ false\ then\ 0\ else\ 1$ in the environment $\eta_0 = \lambda y \in Var.type\_error$

b) Suppose e_1 and e_2 are expressions with $V[\![e_1]\!]\eta = integer$ and $V[\![e_2]\!]\eta = boolean$ in every environment. Show how to calculate the meaning of the expression $let\ x{:}\ int = e_1\ in\ (if\ e_2\ then\ e_1\ else\ x)$ in environment $\eta_0 = \lambda y \in Var.type\_error$.

c) In declaration let $x{:}\ \tau = e\ in\ e$, the type of x is given explicitly. It is also possible to leave the type out of the declaration and infer it from context. Write a semantic clause for the alternative form, $let\ x = e1\ in\ e2$ by using the following partial solution (i.e., fill in the missing parts of this definition):

$$V[\![let\ x = e_1\ in\ e_2]\!]\eta = \begin{matrix} V[\![e_2]\!](\eta[x \rightarrow \sigma])\ \ if \\ type\_error\ \ o.w \end{matrix}$$

Answer. (a)

The integer part of the given condition in the question will be true because of $if\ V[\![false]\!]\eta_0 = V[\![0]\!]\eta_0 = V[\![1]\!]\eta_0 = boolean$ won't be satisfied. The second condition will be true and the result of the evaluation will be integer.

Answer. (b)

The expression result will be $V[\![if\ e_2\ then\ e_1 else\ x]\!](\eta[x \rightarrow integer])$ because the $V[\![e_1]\!]\eta = integer\ and\ \tau = int$. And the evaluation will be reduced to the term $V[\![if\ e_2\ then\ e_1\ else\ x]\!](\eta[x \rightarrow integer])$ which after the evaluation will result in an $integer$.

4.11 Lazy Evaluation and Parallelism

In a "lazy" language, we evaluate a function call f(e) by passing the unevaluated argument to the function body. If the value of the argument is needed, then it is evaluated as part of the evaluation of the body of f. For example, consider the function g defined by

In the let expression, which is a form of local declaration, the type τ may be either int or $bool$. The meaning $V[[e]](\eta)$ of an expression e depends on an environment.

a) Assume we evaluate $g(e_1, e_2)$ by starting to evaluate g, e_1, and e_2 in parallel, where g is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen?
b) Now, suppose the value of e1 is zero and evaluation of e_2 terminates with an error. In the normal (i.e., eager) evaluation order that is used in C and other common languages, evaluation of the expression $g(e_1, e_2)$ will terminate in error. What will happen with lazy evaluation? Parallel evaluation?
c) Suppose you want the same value, for every expression, as lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate $g(e_1, e_2)$ by starting all in parallel, if the value of e_1 is zero and evaluation of e_2 terminates in an error?
d) Suppose now that the language contains side effects. What if e_1 is z and e_2 contains an assignment to z. Can you still evaluate the arguments of $g(e_1, e_2)$ in parallel? How? Or why not?

Answer. (a)

We have multiple processes to evaluate these three subexpressions in parallel. So when reaching the section which is responsible for evaluation of g there will be the need for other subexpressions to be evaluated. We will wait until the evaluation is done and then we'll replace the evaluated value of e_1 or e_2 by themselves in parent expressions.

Answer. (b)

Since the lazy evaluation only concerns about the term which is being used lazily like x in the first if statement. So the whole g function shall not be terminated but remember that we have defined this condition only and only if we know that x is equal to zero. What happens if x was not equal to zero and y needed to be evaluated correctly. Then the g had to be terminated completely.

Answer. (c, d)

So, in a parallel runtime system, we ought to implement some kind of message passing system in order to transfer the result of evaluated variable(statements) to the other dependent process in other parts of the runtime system. While implementing the laziness concept inside the hardware, we need to consider the places that laziness might improve the problem and ignore the terminate messages from other cores (other locations of runtime system). But, in a non-lazy system, the termination in one core results in the termination of all cores.

4.12 Single-Assignment Languages

A number of so-called single-assignment languages have been developed over the years, many designed for parallel scientific computing. Single-assignment conditions are also used in program optimization and in hardware description languages. Single-assignment conditions arise in hardware as only one assignment to each variable may occur per clock cycle.

a) Explain how you might execute parts of the sample program

in parallel. More specifically, assume that your implementation will schedule the following process in some way:

```
x = 5;
y = f(g(x),h(x));
if y==5 then z=g(x) else z=h(x);
```

For each process, list the processes that this process must wait for and list the processes that can be executed in parallel with it.

```
process 1 - set x to 5
process 2 - call g(x)
process 3 - call h(x)
process 4 - call f(g(x),h(x)) and set y to this value
process 5 - test y==5
process 6 - call g(x) and then set z=g(x)
process 7 - call h(x) and then set z=h(x)
```

b) If you further divide process 6 into two processes, one that calls $g(x)$ and one that assigns to z, and similarly divide process 7 into two processes, can you execute the calls g(x) and h(x) in parallel? Could your compiler correctly eliminate these calls from processes 6 and 7?
c) Would the parallel execution of processes you describe in parts (a) and (b), if any, be correct if the program does not satisfy the single-assignment condition?
d) Is the single-assignment condition decidable?
e) Suppose a single-assignment language has no side-effect operations other than assignment. Does this language pass the declarative language test? Explain why or why not?

Answer (a)

The objectives of each process will be as the following in each case:

$$p_1\ set\ x\ to\ 5$$
$$p_2\ call\ g(x)$$
$$p_3\ call\ h(x)$$
$$p4\ \ call\ f\big(g(x), h(x)\big)\ and\ set\ y\ to\ this\ value$$
$$test\ y == 5$$
$$call\ g(x)\ and\ then\ set\ z = g(x)$$
$$call\ h(x)\ and\ then\ set\ z = h(x)$$

So the status of the system after running these processes will be:

$$p_1 \rightarrow no\ wait\ needed$$
$$p_2 \rightarrow wait\ for\ p_1$$
$$p_3 \rightarrow wait\ for\ p_2$$
$$p_4 \rightarrow no\ wait\ needed$$
$$p_5 \rightarrow test\ y == 5\ and\ wait\ for\ p_4$$
$$p_6 \rightarrow no\ wait\ needed(in\ this\ case)$$
$$p_7 \rightarrow no\ wait\ needed(in\ this\ case)$$

Answer (b)

Here is the extended version of the above process list in order to evaluate the assignments inside the two last processes in a new process.

$$p_6\ will\ call\ the\ g(x)$$
$$p_7\ \ z = g(x)$$
$$p_8\ will\ call\ the\ h(x)$$
$$p_9\ z = h(x)$$

The above extended process list simply shows that the $p_6\ and\ p_8$ can be executed in parallel but in the location where assignment to z occurs, we'll consider a practical solution for handling the two-assignment problem.

Answer (c)

No, it couldn't be correct anymore. We cannot risk distributing the lines of code in which the assignment to the same variable is happening at the same time on different processors (or processes). Unless, a practical solution is considered.

Answer (e)

No, assignment will make expressions dependent to each other. So the above program won't pass the declarative test.

4.13 Functional and Imperative Programs

Many more lines of code are written in imperative languages than in functional ones. This question asks you to speculate about reasons for this.

a) Are there inherent reasons why imperative languages are superior to functional ones for the majority of programming tasks? Why?
b) Which variety (imperative or functional) is easier to implement on machines with limited disk and memory sizes? Why?
c) Which variety (imperative or functional) would require bigger executables when compiled? Why?
d) What consequence might these facts have had in the early days of computing?
e) Are these concerns still valid today?

Answer (a)

a) That's not because of the power of imperative languages over the functional ones. They have their own abstraction of computation and can be both used to implement the same tasks. But, a functional programming language, requires a more expressional way of thinking. So there are some radical ways of thinking present in the imperative languages which are not present in functional ones.
b) Imperative languages are easier, because the functional languages will have to obey the rule of immutable data types and will end up with more memory requirements since there are lots of object duplication inside the memory for running a functional program.
c) This cannot be said concretely but, there are programs in imperative languages with multiple lines of code which can be written in only one line of code in functional ones.
d) In the early days of computing there were problems like lack of memory, slow processors and etc. and because of that, there wouldn't be much more room for functional programming to shine.
e) These days the raw processing power of every single personal computer can beat supercomputers of those days. Considering that in mind, there will be a lot room for functional programming to shine. There is also a better optimization of garbage collection and memory management is being done. So there wouldn't be that much problems in practice.

4.14 **Functional Languages and Concurrency**

It can be difficult to write programs that run on several processors concurrently, because a task must be decomposed into independent subtasks and the results of subtasks often must be combined at certain points in the computation. Explain why functional programming languages do not provide a complete solution to the problem of writing programs that can be executed efficiently in parallel. Include two specific reasons in your answers.

Answer

There are two vast principles in concurrent programming:

1) Multiprogramming
2) Multiprocessing

Unfortunately, there haven't been a complete solution to the multiprocessing with 100 percent utilization. And there is a major need to the context and process switching

in a multiprogramming environment. The vision of functional programming languages enforces this situation.