e-ISBN : 978-629-98755-1-2

SIG : e-Learning@CS
https://appspenang.uitm.edu.my/sigcs/
Publication Date : 5 – Apr - 2024

FUNCTIONAL PROGRAMMING PARADIGM WITH SCHEME PROGRAMMING LANGUAGE

*Jamal Othman[1]
*jamalothman@uitm.edu.my[1]

[1]Jabatan Sains Komputer & Matematik (JSKM),
Universiti Teknologi MARA Cawangan Pulau Pinang, Malaysia

*Corresponding author

ABSTRACT

Functional programming (FP) is a paradigm which the expression is written in declarative style or bind the expression as mathematical function. FP treats functions as data. Basically, this paradigm was introduced for mathematical computation. Anything that can be computed by the FP than it is considered as computable. Currently, this paradigm has been introduced as an elective or optional course to the students at the tertiary level of education. Other than FP paradigms, the students are also introduced with the structured, object-oriented, logic and scripting paradigms. The main purpose of introducing varieties of programming paradigms is to make sure that the students are able to choose appropriate programming language related to their project scopes and domain. The FP paradigms focus on what is the expected result the program should produce rather than on how the result will be get as applied in structured and object-oriented programming paradigms. This article will discuss details on the characteristics, example of codes which uses the Scheme programming language and implementation of the FP paradigms in the real life.

Keywords: functional programming (FP), paradigms, scheme, lambda calculus

Introduction

Functional programming (FP) is based on lambda calculus which was developed by Alonzo Church in 1930s, for studying computations with functions (Bhadwal, 2022). The coding in FP is a declarative type that is focusing on what to solve instead of on how to solve. The function is the main element in the FP, similarly as object becomes the main tools in the object-oriented programming (Vishal, 2022). Examples of programming languages that support the FP paradigms are Haskell, JavaScript, Python, Scala, Erlang, Lisp, ML, Clojure, OCaml, Common Lisp, Scheme and Racket.

One of the uniqueness of FP is the implementation of recursive functions to avoid the common repetition control structures such as the `for` loop, `while` or the `do..while` loops as implemented in imperative programming paradigms. FP applies the immutable data approach which the data state cannot be modified or changed after it is created. The traditional approach of programming paradigm such imperative or structured programming applied the mutable data approach which the code will overwrites the old data whenever the new data is available. FP paradigm supports the parallel programming and concurrency for multilayer computations (Khanfor & Yang, 2017). Moreover, FP diverges from the practice of relying on the sequence of codes for application execution, a characteristic seen in imperative or object-oriented programming paradigms (Parewa, 2022).

e-ISBN : 978-629-98755-1-2

SIG : e-Learning@CS
https://appspenang.uitm.edu.my/sigcs/
Publication Date : 5 – Apr - 2024

Characteristics of Functional Programming (FP)

FP consists of predefined or user-defined functions. Each function will be defined according to the given expression. Expressions will be formed to construct a special function which consists of other functions as substitution function, variables and constant values. Every expression should be represented by certain values and the computation will be done to determine the results (Chitil, n.d.). Computation in FP using the **Cambridge Prefix notation** as shown in table 1 below.

Table 1: Computation of expression in imperative vs FP

Imperative paradigm (Infix expression)	**Functional paradigm (Prefix notation)**
`5 + 4 + 4 + 3`	`+ 5 4 4 3`
`5 − 4 + 3`	`+ − 5 4 3`
`(9 + 4) * (5 − 2)`	`(* (+ 9 4) (− 5 2))`

FP has a special feature which allow the user to delay the processing or computation. This feature is called as **lazy evaluation**. Lazy evaluation is defined as the expressions will be evaluated whenever it is actually needed or required only (GNU, n.d.). The following table 2 shows the difference of eager evaluation and lazy evaluation as applied in functional paradigm by using the Scheme programming language.

Table 2: comparison of eager and lazy evaluation

(Eager evaluation)	**(Lazy evaluation)**
(define (eager x y)(+ (expt x 2) (expt y 2))) > (eager 6 8) 100 ; the expressions will be evaluated immediately whenever the value is given.	(define (lazy x y) (delay (+ (expt x 2)(expt y 2)))) > (lazy 7 8) #<promise:lazy> ; no result shown here > (force (lazy 7 8)) 113 ; the result shown here after forced ; the command ‘`delay`’ delayed the expressions evaluation. It will be processed whenever it is needed by using the command ‘`force`’ for immediate computation.

FP is actually based on the **lambda calculus** which in turn provides a framework for studying decidability questions of programming (Aaby, 1998). The function can be created using the Scheme programming language either by implementing or not implementing the keyword ‘lambda’. The following table 3 shows the definition of Pythagorean Theorem.

Table 3: Pythagorean Theorem with Lambda or without Lambda command

<table>
<tr><th>Without Lambda</th><th>With Lambda</th></tr>
<tr><td><pre>
(define (pythagorean a b)
  (sqrt (+ (expt a 2) (expt b 2))))

(display " Enter the value of a ")
(define a(read))
(display " Enter the value of b ")
(define b(read))
(define result(pythagorean a b))
(display " The result is ")
(newline)
(display result)

Output:
Enter the value of a 6
Enter the value of b 8
The result is
10
</pre></td><td><pre>
(define pythagorean (lambda(a b)
  (sqrt (+ (expt a 2) (expt b 2)))))

(display " Enter the value of a ")
(define a(read))
(display " Enter the value of b ")
(define b(read))
(define result(pythagorean a b))
(display " The result is ")
(newline)
(display result)

Output:
Enter the value of a 6
Enter the value of b 8
The result is
10
</pre></td></tr>
<tr><td>The name of the function is pythagorean and sends two parameters, a and b.</td><td>The name of the function is pythagorean and the lambda function sends two parameters, a and b.</td></tr>
</table>

FP allows us to store multiple data or arguments in a list (Othman et. al, 2019). Additionally, the commands `cons` can be used to construct pairs and pairs are used to construct lists. The `car` and `cdr` commands enable us to retrieve the first element or argument from the list and extract the remainder arguments except the first argument of the list respectively. Figure 1 shows the implementation of the commands `list`, `cons`, `car` and `cdr` in Scheme programming language.

```
(list '(ali abu ibrahim jusoh aminah raju kamilia))
> ((ali abu ibrahim jusoh aminah raju kamilia)) ; → list contents
(cons '(ali abu) '(ibrahim jusoh aminah raju kamilia))
> ((ali abu) ibrahim jusoh aminah raju kamilia) ; → contents in the list by using the cons command

(car '(ali abu ibrahim jusoh aminah raju))
> Ali ; → the first element from the list

(cdr '(ali abu ibrahim jusoh aminah raju))
> (abu ibrahim jusoh aminah raju); → the remainder elements from the list except the first element

> (define flower '(rose tulip carnation chrysanthemum orchid))
> flower
(rose tulip carnation chrysanthemum orchid) ; → flower is a list which consist name of flowers
> (car flower)
rose; → first element of the flower lists.
> (cdr flower)
(tulip carnation chrysanthemum orchid); → the remainder elements of the flower lists
```

Figure 1: implementation of list, cons, car & cdr commands in Scheme programming language

Similar to other type of programming paradigms, the FP is also providing the selection control structures such as the `cond` or `case` for multiple conditions and `if` for single or dual conditions (Othman et. al, 2019). The following table 4 shows the implementation of selection control structures in Scheme programming language as compared to imperative paradigm and the output as shows in table 5.

Table 4: Implementation of selection control structures in FP as compared to Imperative paradigm.

Imperative Paradigms with C

```c
#include <stdio.h>
float rateFunct(char);
float bonusFunct(float);

int main()
{
            char workertype;
            float rateperday, totalsalary, bonus, totalall;
            int numberofworkingdays;

            printf("\n Payroll System ");
            printf("\n Worker Type ");
            printf("\n A. Senior Manager ");
            printf("\n B. Manager ");
            printf("\n C. Supervisor ");
            printf("\n D. Production line worker ");
            printf("\n ? ");
            scanf(" %c", &workertype);
            printf("\n Number of working days : ");
            scanf("%d",&numberofworkingdays);

            rateperday = rateFunct(workertype);
            totalsalary = rateperday * numberofworkingdays;
            totalall =  totalsalary + bonusFunct(totalsalary);

            printf("\n Salary RM %.2f ",totalsalary);
            printf("\n Bonus RM %.2f ",
                                    bonusFunct(totalsalary);
            printf("\n Salary+Bonus RM %.2f ",totalall);
            printf("\n Thank you ");

            return 0;
}

float rateFunct(char workertype)
{           float rateperday;
            if (workertype == 'A' or workertype == 'a')
                        rateperday = 200;
            else if (workertype == 'B' or workertype == 'b')
                        rateperday = 150;
            else if (workertype == 'C' or workertype == 'c')
                        rateperday = 100;
            else if (workertype == 'D' or workertype == 'd')
                        rateperday = 75;
            else
                        rateperday = 0;
            return rateperday;
}

float bonusFunct(float totalsalary)
{           float bonus;
            if (totalsalary > 3000)
                        bonus = 300;
            else
                        bonus = 150;
            return bonus;
}
```

Functional Paradigms with Scheme

```scheme
#lang scheme

(define (get-rate)
  (display "Payroll System")
  (newline)
  (display "Worker type ")
  (newline)
  (display "A. Senior Manager ")
  (newline)
  (display "B. Manager ")
  (newline)
  (display "C. Supervisor ")
  (newline)
  (display "D. Production line worker ")
  (newline)
  (display "?")
  (let ((code (read)))
    (newline)
    (cond
     ((or (eq? code 'A)(eq? code 'a)) 200)
     ((or(eq? code 'B)(eq? code 'b)) 150)
     ((or(eq? code 'C)(eq? code 'c)) 100)
     ((or(eq? code 'D)(eq? code 'd)) 75)
     (else 0))))

; another option to apply the case control structure
  ; as shown below
   (case code
     ((A a) 200)
     ((B b) 150)
     ((C c) 100)
     ((D d) 75)
     (else 0))))

(define (get-workingdays)
  (display "Number of working days :")
  (read) )

(define (main)
  (let* ((rate (get-rate))
         (numberofworkingdays(get-workingdays))
         (tot-salary (* rate numberofworkingdays))
         (bonus (if (>= tot-salary 3000) 300 150))
         (tot-all (+ tot-salary bonus)))
    (display "Salary RM ")
    (display tot-salary)
    (newline)
    (display "Bonus, RM ")
    (display bonus)
    (newline)
    (display "Salary+Bonus, RM ")
    (display tot-all)
    (newline)
    (display "Thank you")
 ))

(main)
```

Table 5: The output of selection control structures in FP as compared to Imperative paradigm.

Imperative Paradigms with C	Functional Paradigms with Scheme
Output : Payroll System Worker Type A. Senior Manager B. Manager C. Supervisor D. Production line worker ? A Number of working days: 30 Salary RM 6000 Bonus RM 300 Salary+Bonus, RM 6300 Thank you	Output : Payroll System Worker type A. Senior Manager B. Manager C. Supervisor D. Production line worker ? A Number of working days: 30 Salary RM 6000 Bonus, RM 300 Salary+Bonus, RM 6300 Thank you

Functional paradigms in Scheme programming language provides a predicate type which it is a built-in procedure that always returns the boolean value (`#t` or `#f`) (Racket, n.d.). The following table 6 shows the lists of built-in procedures with predicate type in Scheme programming language.

Table 6: List of Built-in Procedures with Predicate type in Scheme Programming Language

Predicate Type	Function or purposes	Example

> (procedure? f1)	To examine the existence of the function name	`> (define add (lambda (x y) (+ x y)))`
		`> (procedure? add)`
		`#t`

> (null? mylist)	To examine the list is empty or not empty	`> (define mylist '(a b c d e))`
		`> (null? mylist)`
		`#f`
		`> (define nextlist '())`
		`> (null? nextlist)`
		`#t`

> odd?
> even?
	To determine either the number is even or odd number	`> (define x 10)`
		`> (even? x)`
		`#t`

> boolean?	To determine either the expression is true or false	`> (boolean? (> 9 3))` `> #t`
> negative? **> positive?**	To determine the expressions or value is a negative of positive value	`> (positive? (- 10 -12))` `#t`

> eq?	To determine the value is similar with the value assign to another identifier	`> (define option1 'A)`
		`> (define option2 'a)`
		`> (define option3 'A)`
		`> (eq? option1 option2)`
		`#f`
		`> (eq? option1 option3)`
		`#t`

**only the most frequently used of built-in predicate are listed here*

Scheme has no expressions designed for looping. The only easy way to do this is recursion, that is, designing a procedure such that it meets 2 criteria which the procedure must have a base case that it stops at and the recursive function. Recursive is a function or procedure that calls itself. In Scheme programming language, simple code of iteration can be achieved through recursion by having a function that call itself. Most programs are tail recursive, where the recursive function calls the last action that occurs. In other words, there is no need to return for further execution of the n-th iteration of the function after the recursive function calls the (`i+1`) iteration (Southwestern University, n.d). The following table 7 shows the implementation of recursive function in C and Scheme programming language for Fibonacci problems.

Table 7: Recursive function in C and Scheme Programming language

Imperative paradigm (C programming)	Functional paradigm (Scheme Programming)

#include <stdio.h>
int fibonacci(int);
int main()
{
int number, result;
printf("\n Enter a number : ");
scanf("%d", &number);
 	
result = fibonacci(number);
 	
printf("\n The fibonacci number for %d is %d ", number, result);
 	
return 0;
}
 	
int fibonacci(int x)
{
if (x == 0)
return 0;
else if (x == 1)
return 1;
else
return (x + **fibonacci(x-1)**);
}
 	
Output :
Enter a number : 5
The fibonacci number for 5 is 15
	#lang scheme
	(define fibonacci (lambda (x)
	(cond ((eq? x 0) 0)
	((eq? x 1) 1)
	(else **(+ x (fibonacci (- x 1)))**)
	)
	)
	)
	
	(display "Enter a number : ")
	(define x(read))
	(newline)
	(define fibo(fibonacci x))
	(display "The fibonacci number for ")
	(display x)
	(display " is ")
	(display fibo)
	
	Output :
	Enter a number : 10
	The fibonacci number for 10 is 55

Functional Programming (FP) Applications

Quite rare actually we heard or found the commercial application systems which implements the FP paradigms. FP is not the most common paradigm, and most developers are not intimately familiar with its features and syntaxes. Another main reason is the use of recursion structure instead of ordinary loops in the making of the application makes most of the developers are refused to use the FP paradigm. Parallelism processing is the most demanding field which are now increasingly applied in industrial and commercial area. The strength of parallelism tools to solve the concurrency issues in FP, has developed the consciousness and confidence among developers to use FP paradigm.

Nowadays, FP applications appear in diverse fields such as complex networking switches, event correlation managers, expert contract valuators, integrated circuit designers, theorem provers & model checkers, natural language processors and robotics and manufacturing (ByteScout, n.d.). First FP success story begins when one of the largest global manufacturers of telecommunications equipment, Ericsson which operates in more than 100 countries and 80,000 employees, uses the Erlang FP language in a variety of telecommunications and networking devices. Applications developed for this equipment prove highly reliable with only a few seconds of downtime over the course for many years. The second FP application is the chip design assistant. Bluespec is a commercial company has claimed that the development of their chip design assistant platforms derives from the Haskell FP language. The third example of FP application is Jane Street Capital located in US is a proprietary trading firm involved in financial markets around the world used the FP paradigm to develop sophisticated statistical research operating over terabytes of data as well as real-time systems that demand performance (Wadler, n.d.).

Conclusion

In conclusion, FP is a paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It emphasizes immutability, lazy evaluation, and the use of higher-order functions. FP languages, such as Haskell, Lisp, and Scheme, have increased popularity for their ability to improve source code simplicity, modularity, and maintainability. Moreover, functional programming aligns well with modern trends in software development, such as the rise of distributed systems and the increasing importance of parallel processing in a world of multi-core processors. As the industry continues to evolve, functional programming concepts are likely to become even more relevant, influencing not only specialized functional languages but also mainstream languages that adopt functional features. Ultimately, the adoption of functional programming is a matter of choosing the right tool for the task at hand. It may not be suitable for every project, but incorporating functional programming principles into one's coding practices can lead to more robust, modular, and maintainable software systems.

References:

Aaby, A. A. (1998, January 1). Functional Programming. Https://www.cs.jhu.edu/. Retrieved November 12, 2023, from https://www.cs.jhu.edu/~jason/465/readings/lambdacalc.html#:~:text=Functional%20programming%20is%20based%20on%20the%20lambda%2Dcalculus%20which%20in,programs%20into%20equivalent%20functional%20programs.Bhadwal, A. (2022, September 9). Functional Programming Languages: Concepts & Advantages. Hackr.io. Retrieved November 10, 2023, from https://hackr.io/blog/functional-programming

ByteScout Team of Writers (n.d.). REAL-WORLD SUCCESS STORIES IN FUNCTIONAL PROGRAMMING. ByteScout. Retrieved November 14, 2023, from https://bytescout.com/blog/functional-programming.html

Chitil, O. (n.d.). Functional Programming. Kent Academic Repositity. Retrieved November 11, 2023, from https://kar.kent.ac.uk/24064/1/FuncOlaf.pdf

GNU (n.d.). 17. Lazy evaluation. T.Shido's Home Page. Retrieved November 11, 2023, from https://www.shido.info/lisp/scheme_lazy_e.html

Khanfor, A. & Yang, Y. (2017). An Overview of Practical Impacts of Functional Programming. 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), 50-54. https://doi.org/10.1109/APSECW.2017.27

Othman, J., Ahmad, J.I., Abdul Wahab, N., Che Jan, N.Y., & Abd Wahab, Z.I. (2019), Programming Paradigms Concepts, (First ed.), Selangor, Malaysia: Penerbit UiTM, ISBN: 978-967363590

Parewa Labs Pvt. Ltd. (2022, January 1). What is Functional Programming? A Beginner's Guide. Programiz. Retrieved November 9, 2023, from https://programiz.pro/resources/what-is-functional-programming/

Racket. (n.d.). Predicates. Functional Programming With Scheme Programming Language. Retrieved November 13, 2023, from https://docs.racket-lang.org/predicates/index.html#%28def._%28%28lib._predicates%2Fmain..rkt%29._if~3f%29%29

Southwestern University (n.d.). Predicates. Scheme Recursion/Lambda Lab. Retrieved November 13, 2023, from https://people.southwestern.edu/~owensb/PL/RecursionLambdaLab.htm#:~:text=Recursion%20is%20a%20term%20used,the%20last%20action%20that%20occurs.

Vishal (2022, June 28). Functional Programming Paradigm. GeeksForGeeks. Retrieved November 9, 2023, from https://www.geeksforgeeks.org/functional-programming-paradigm/

Wadler, P. (n.d.). Functional Programming in the Real World. Https://Homepages.inf.ed.ac.uk/. Retrieved November 14, 2023, from https://homepages.inf.ed.ac.uk/wadler/realworld/