Annales Mathematicae et Informaticae
36 *(2009) pp. 163–174*
http://ami.ektf.hu

Teaching programming language in grammar schools

Zoltán Hernyák, Roland Király

Eszterházy Károly College, Department of Information Technology

Submitted 6 October 2009; Accepted 10 December 2009

Abstract

In Hungary algorithmical thinking is a part of teaching informatics both in primary and secondary grammar schools. A teacher usually starts with some everyday algorithm, taking examples from cooking or solving a mathematical or physical problem. The steps of the solutions are usually represented in a flow diagram. This diagram is a graphical representation of the algorithm steps including decision symbols. With these decision symbols, selections and iterations can be applied.

Unfortunately, the common ways of describing algorithms are far from functional thinking, therefore it is rather difficult for teachers to find materials on teaching functional programming. On the other hand, programming and trying the algorithm in a functional way is much easier as in the imperative way ([3], [4]).

The next step is usually the description of the algorithm by a sentence-like language, which is very close to BASIC programming language. At this point the teacher switches to a programming language, like Pascal, BASIC, C# [17], or any OOP [9] supportive or OOP language [9], all of which are imperative languages.

These languages were taught to teachers during their studies, and are used in their workplaces, in grammar schools as well. We believe that the functional programming paradigm is raising nowadays, and getting more and more important. In this paper we are trying to show and prove that this programming style is appropriate for teaching programming in grammar schools.

1. Introduction

The mathematical fundamentals of functional programming are based on Church's lambda-calculus theory which he invented in 1932–33. Turing proved that the effectively evaluable functions interpreted on non-negative integers following the lambda-calculus are the same as the ones that are computable by the Turing machine commonly utilized by imperative languages. Thus every task which can be solved by imperative languages are as well solvable in functional languages, and vice versa.

There are several functional languages, which can be used to code the algorithms. Most of them can be downloaded and use for free, and has some sort of IDE (Integrated Development Environment).

Erlang is a development of Ericson and Ellemtel Computer Science Laboratories. Erlang is a programming language in that it is possible to develop concurrent, real-time, distributed and highly error-tolerant systems. Ericson uses the Open Telecom Platform extension of Erlang to develop telecommunication systems. The language has internal methods to achieve that without shared memory distributed applications communicate through signaling among themselves. It supports integrating components written in different programming languages, but is generally a weakly typed language.

Haskell is an advanced purely functional programming language. An open source product of more than twenty years of cutting edge research, it allows rapid development of robust, concise, correct software. With strong support for integration with other languages, built-in concurrency and parallelism, debuggers, profilers, rich libraries and an active community, Haskell makes it easier to produce flexible, maintainable high-quality software.

Clean is a non-profit development as a functional programming language, with many similarities to Haskell. With the ObjectIO library extension of Clean, one can develop interactive applications having menus, and dialog windows.

In Hungary at Eötvös Lóránd University the functional programming paradigm is used both in education and in scientific researches and projects. There are attempts to include functional languages in education abroad too.

We use Clean functional programming language as a reference language to solve some basic algorithmic problems. The Clean System is a software development system for developing applications in Clean. The Clean System is available on many platforms (PC, Mac, SUN) and operating systems (Windows'95/'98/2000/NT, Linux, MacOS, Solaris). The main platforms are PC and Mac. The Clean System is a full-fledged system that can be used in industrial environments. The Clean System is a commercial product of Hilt–High Level Software Tools B.V. Clean can be downloaded from its homepage, `http://clean.cs.ru.nl/index.html`.

We do not intend to give a full language reference here, as many information is available in [5] and [15]. We give just a short introduction on how this language (and other functional languages) can be used as a reference language in teaching the implementation of simple algorithms.

Short introduction to Clean language. First we show how simple the usual Hello World! program (see Example 1) is. A one-module Clean program starts with the keyword "module" which is followed by the module name. It must be equivalent with the file name, so if the module name is *helloWorld*, the source code must be saved to *helloWorld.icl*. The second line imports the StdEnv (standard environment), which holds the prototypes of the most important library functions and type definitions.

The Start expression is the replacement of the C-like main function. The evaluation of the Start expression creates the result of the functional program itself. The first program demonstrates how simple the Hello World! program is.

Clean source code

```
module helloWorld
import StdEnv

Start = "Hello World!"
```

Example 1: Hello World!

In the following examples the first two lines are not shown, we should concentrate on the significant lines. In example 2 we assign a numerical expression as the result of the program. In a console application this expression is evaluated and written on the screen.

Clean source code

```
Start = 3-2*4
```

Example 2: Numeric expression

In a functional program we do not have the usual variables, but we can use simple constant value functions similarly to the variables, or constants. In Example 3, *a* and *b* are functions, which evaluate to the values *2* and *3*. As local functions, they are defined inside the Start expression's scope, using the *where* keyword.

Clean source code

```
Start = a+b
  where
    a=2
    b=3
```

Example 3: Using functions

The previous examples have demonstrated how easy the very first steps are in the functional world. Not only simple, but also more complicated types can be used as a result of the program. For example we can define lists easily, and use them as result (see Example 4).

Clean source code

```
Start = resultList
  where
    resultList = [1,3,4,5,6,7,9,4,3,5,6,7,8,4,3]
```

Example 4: Define list

A special datatype called tuple can be found in functional languages. Tuples can be imagined as records without defining its naming the fields. Tuples are values keeping together. We can refer to the values of a tuple by their serial numbers only (1st element, 2nd element, etc.). With tuples we can define functions that return more than one value at a time. In example 5 the Start expression returns a tuple of a string and an integer value. Running this program will show both elements on the screen (separating by commas), writing *3+2=,5*.

Clean source code

```
Start = ("3+2=",3+2)
```

Example 5: Returning tuple

Defining a function usually does not require the description of the function type, it is inferred by the deduction system included in the compiler. We must give a name to the parameter and define the result. In this example we define a function called *increment*, which only has one argument, named *a*. We define the result of this function as *a+1*. The type deduction system will know that the type of the *a* can be anything that the additive operator with an integer value can interpret. We call this function in the Start expression, and give the value of *3* to it. The type deduction system will check if its type (integer) can be added to another integer, and will generate an increment function with this specific type.

Clean source code

```
Start = increment 3

increment a = a+1
```

Example 6: User defined function

Note that calling a function means writing its name and after a space, defining the value for its parameter. There's no need for the C-style function to call operators (parentheses). Nor we use parentheses when a parameter value is a complex expression (see Example 7).

Clean source code

```
Start = decrement (increment 3)

increment a = a+1
decrement a = a-1
```

Example 7: Calling a function

In example 7 we want to evaluate the inner expression first (*increment 3*), then pass its value to function *decrement*.

We can use *patterns* to define different function bodies for different input arguments. In this example we decrement every positive value by one, but decrementing zero means returning the zero value itself (see Example 8).

When we write *decrement 0* (it means " if the first parameter's value equals to 0"), we define the function result as 0. In other cases we define the function result

Clean source code
```
decrement 0 = 0
decrement a = a-1
```

Example 8: Using patterns

as *a-1.* Note that adding a negative value to the function will trigger the second variant of the function body, as a negative value is not equal to 0. We can define different cases for negative values, but not with the pattern match (as we cannot write all the patterns for all the negative numbers), but we can write a *guardian* term (see Example 9).

A guard is a boolean expression that can be inserted between the patterns of a function alternative and the symbol =. The symbol | separates the patterns and the guard. The alternative is only applied when the guards yield True. Each function clause can have a sequence of guarded right-hand sides.

This time we first check if the value of n is less than m or not. If less, we define the function result as the value of m, because it is the maximum of the two numbers. In every other case we define the function result as n, as it holds the maximum value.

Clean source code
```
maximum n m
  | n<m        = m
  | otherwise  = n
```

Example 9: Using guard

In example 9 we used the word *otherwise* and it seems as it would be a keyword for these cases, but it is not. The word *otherwise* is a constant function, always returning with the value of *true.* That means we could write the word *True* instead (see Example 10). Note that we can define more than two cases at the same time, writing more guard terms.

Clean source code
```
maximum n m
  | n<m   = m
  | True  = n
```

Example 10: Using True instead of otherwise

Pattern matching can be applied not only to simple values, but also to lists. In example 11 the first pattern matches to the empty list, and returns zero. The second pattern matches to a list containing one value only. In this case, the parameter named x will holds the value of this element. The third pattern will match to a list which has at least one element. The first element's value will be represented by x in this case, the remaining list goes to *tail*, which means that in this case x is a simple element, and *tail* is a list of elements. Note that the last pattern matches to a one-element list too, and in that case *tail* will be an empty list.

Clean source code

```
count [] = 0
count [x] = 1
count [x:tail] = 2
```

Example 11: Patterns of list

We use the colon operator to add (insert) an element to a list. Note that it will not modify the original list, but will create a brand new list, as function side-effects are denied in functional languages. Function *insert* in Example 12 will insert the parameter value of a after the first element of the list by constructing a brand new list.

Clean source code

```
insert a [x:tail] = [x:a:tail]
```

Example 12: Inserting a value into a list

2. Examples of using functional programming methods

Element of a set. Now we will discuss the basic algorithm of *determining if a given value is an element of a set or not.* The set is given as a list of integer values. Normally it is given in imperative algorithm in 13.

Imperative algorithm

```
algorithm isElement
  parameters x:integer, h:list of integer
start
  i:=1
  while i<=length of h and h[i]<>x
     i  := i+1
  end of while
  return (i<=length of h)
end of algorithm
```

Example 13: isElement as imperative algorithm

If the functional paradigm is used to write the previous program, the function can be evaluated in two ways. The first clause is evaluated if the set is empty, so value x cannot be element of this empty set. In the second case we separate the set into two parts, an element of the set, and the remains of the set (tail). We can say that x is an element of this set, if it equals to the separated element of the set, or if it is an element of the remaining set (see Example 14). We can use this *isElementOf* function as in the example 15.

Clean source code
```
isElementOf x [] = False
isElementOf x [a:tl]
 | x==a         = True
 | otherwise  = isElementOf x tl
```

Example 14: isElementOf function

Clean source code
```
Start = isElementOf  3 mySet
  where
    mySet  = [1,2,3,5,6,8]
```

Example 15: Use of isElementOf

Counting of elements. Counting the elements can be carried out similarly to the exampe above. The empty list has zero elements, in other cases we can define the length of the list by counting one element at a time (see Example 16).

Clean source code
```
countingElements []  = 0
countingElements [x:tl] = 1 + countingElements tl
```

Example 16: countingElements function

When we want to count those elements only that have a *P property* (in this example, the elements that are *even*), we should modify this function a little by introducing a guard expression. With that we can separate two cases: whether the first element of the list has P property or not (see Example 17).

Clean source code
```
countingElementsEvens []  = 0
countingElementsEvens [x:tl]
  | isEven x   = 1 + countingElementsEvens tl
  | otherwise = countingElementsEvens tl
```

Example 17: counting of P property

Note that the *isEven* function is a library function, and its parameter type must be Int, and its result has to be a boolean value (see Example 18). In Clean, however, we do not need to define the type of a function, in simple cases the type inference system will deduce that. If we want to define a function type explicitly, we can use the double colon after which we can list the types of the input parameters. After the arrow we can give the result type.

Clean source code
```
isEven::Int->Bool
```

Example 18: Prototype of isEven

We can define our own function that has one int parameter only and results a bool, too. In example 19 we define an *isGood* function, which checks if the parameter is in range of $[4 \dots 8)$.

```
Clean source code
isGoodElement::Int->Bool
isGoodElement x = (4<x) && (x<8)
```

Example 19: User defined isGoodElement function

Fortunately, functions in Clean can be passed easily as a parameter, if their names are given. We can define the counting of elements algorithm by using the *P property function as a parameter.* If we want to define the type of *countingElementsAny* function, the first parameter is a list of integers, the second is a function which needs one integer, and returns bool.

```
Clean source code
countingElementsAny:: [Int] (Int->Bool) -> [Int]
countingElementsAny []  isP = 0
countingElementsAny [x:tl] isP
  | isP x         = 1 + countingElementsAny tl isP
  | otherwise = countingElementsAny tl isP
```

Example 20: Defining countingElementsAny function

Giving a function as a parameter is very simple, but its type matches only with one parameter. Without explicitly defining the type of *countingElementsAny*, the type inference system will deduce the same. In the Start expression we can call this function by giving a list and a function as a parameter (see Example 21).

```
Clean source code
Start = countingElementsAny myList isGoodElement
  where
    myList = [1,3,4,5,6,7,9,4,3,5,6,7,8,4,3]
```

Example 21: Calling the countingElementsAny general function

Index of an element. Suppose to have a specific value and a list of values. We need to know what the index of the specific value inside the list is. If it is not in the list, the function must return with 0.

In the first case, the element cannot be found in the empty list, so it returns with 0. In the second case, if the first element equals to the given one, we have its index, and it can return it. Otherwise, we try to determine the index of the value in the remaining list (tail). If we found a good index value (other than zero), we must increase that with 1, because we removed the first element of the tail, and therefore the indices in the tail are shifted by one. If we cannot found the element in the tail, we return with 0 as well (see Example 22).

```
Clean source code
indexOf e [] = 0
indexOf e [x:tl]
  | e==x          = 1
  | index>0       = 1+index
  | otherwise     = 0
  where
     index = indexOf e tl
```

Example 22: Calling the countingElementsAny general function

The maximum of elements. Let's suppose we have a list of integers (a set of integers), and need to determine their maximum value. We give a possible solution for this problem as the *myMaxList* function in example 23. We have chosen this name because a *maxList* function exists in the StdEnv standard library.

```
Clean source code
myMaxList [e]   = e
myMaxList [e:tl]
  | e>max     = e
  | otherwise = max
  where
     max = myMaxList tl
```

Example 23: myMaxList

Sum of elements. Let's suppose we have a list of elements, and we have to determine the sum of these elements. We can define a *genSum* function, which takes a list of integers, and generates the sum of the elements recursively as we show in example 24.

```
Clean source code
genSum [] = 0
genSum [x:tl] = x + genSum tl
```

Example 24: Clean program

Selecting of elements. Let's suppose we have a list of elements, and we need the sublist of the values, gathering the ones with a P property. Let's say we have an *isP* function which can decide whether an element has a P property or not. The solution is very similar to the *countingOfElementsAny* (see in example 20). An empty list has no elements with P properties. Otherwise, if the first element has P property, we will insert it into result before the remaining selected elements, or else it returns the selected elements of the tail (see Example 25).

Clean source code
```
selectingElementsAny []  isP = []
selectingElementsAny [x:tl] isP
  | isP x         = [x : selectingElementsAny tl isP]
  | otherwise = selectingElementsAny tl isP
```

Example 25: Selecting elements

Merging two lists into one. Let's suppose we have two ordered lists, and we have to merge them into one list, keeping the ordering as well. The solution in example 26 handles two different cases. When one of the lists is empty, the result is the another (possibly not the empty) list. Otherwise, when either lists are not empty, we can take the first element of both lists, and decide which is less. If the first element of the first list is the least (named x in the function), insert it into the beginning of the result, and process the remaining lists. The same is the case when the first element of the second list is less.

Clean source code
```
merging [] b  = b
merging a []  = a
merging [x:xtl] [y:ytl]
  | x<y         = [x : merging xtl [y:ytl]]
  | otherwise  = [y : merging [x:xtl] ytl]
```

Example 26: Merging elements

Intersect of two sets. Let's suppose, we have two sets, and we have to determine the intersection of the two sets. We can use the *isElement* function defined above, and a solution is given in example 27.

Clean source code
```
intersect [] _ = []
intersect [x:tl] b
 | isElement x b    = [x : intersect tl b]
 | otherwise    = intersect tl b
```

Example 27: intersection of two sets

The underscore sign in the pattern matches to any value (like the joker char matches to any file name). In this case, we can interpret the first pattern as follows: if the first argument is an empty list, the second argument can be anything. This function can be called from anywhere, as described in example 28.

Quick Sort. We can define the Quick Sort algorithm as in example 29. We use a special list construction mode, which is very close to the mathematical way of giving a set. The `[ x \\ x <- r | x<e ]` means: construct a list of elements x, where x comes from a list named r, and x is less than the value of e. The `++` operator concatenates two lists together.

Clean source code
```
Start = intersect set1 set2
  where
    set1 = [1,2,3,5,6,8]
    set2 = [1,3,4,5,7,8,9]
```

Example 28: Calling intersect

Clean source code
```
qsort [] = []
qsort [e:r] = qsort [ x \\ x <- r | x<e ]
              ++ [e] ++
              qsort [ x \\ x <- r | x>=e ]
```

Example 29: qsort function

3. Conclusion

The most important element of the functional languages is the function. As all functional programs are built up of the composition of functions, it is simple to write example algorithms with the help of functions. Experience shows that students have a strong indisposition for using functions, list expressions or pattern matching. The reason for this is that their way of thinking is based on imperative grounds, and that technology is averse in imperative programs. As the use of guards have grounds in imperative paradigms, they can be easily substituted by a kind of *switch* control structure.

When writing simple functions, the variables, or rather the iterations are missing from the imperative way of thinking. Despite considering them nice and elegant, students do not like using recursive solutions because of their difficulty level. This is so, because they use the already acquired imperative solutions as a starting point, and cannot replace iterations with recursive functions. Most undergraduates find the use of function parameters very exciting, and discover their advantages soon.

After acquiring the functional programming technology, they can view and use algorithms on a higher level of abstraction. This has an impact on their imperative programming style and development. The use of recursive functions often causes problems for beginner programmers, because their training in that field is insufficient. They rarely come across functional thinking in other fields or subjects, like mathematics.

They can hardly get used to regarding a function as a complete prototype while writing it. At the same time, these properties of the functions drive them to learn, as they know and feel that the solution is simple, and they know the principle (the imperative algorithm). They work assiduously on their ideas because they know that they will succeed.

References

[1] Joosten, S., Berg, K., Hoeven, G., *Teaching functional programming to first-year students* Journal of Functional Programming (1993), 3:49–65 Cambridge University Press Cambridge University Press 1993 doi:10.1017/S0956796800000599.

[2] Thompson, S., Wadler, P., *Functional programming in education? Introduction* Journal of Functional Programming (1993), 3:3–4 Cambridge University Press Cambridge University Press 1993 doi:10.1017/S0956796800000563.

[3] Járdán T., Pomaházi S., *Adatszerkezetek és algoritmusok* Liceum Kiadó.

[4] Csőke L., Garamhegyi G., *Adatszerkezetek és algoritmusok* Liceum kiadó.

[5] Nyékiné Gaizler J. (szerk), *Programozási Nyelvek*, Kiskapu Kft, 2003.

[6] Barendregt, H.P., *The lambda Calculus, its Syntax and Semantics*, Amsterdam, North-Holland, 1984.

[7] Csörnyei Zs., *Lambda kalkulus - előadás jegyzet (kézirat)*. `http://people.inf.elte.hu/csz/lk-jegyzet.html`.

[8] Horváth, Z., Kozsik, T., *Teaching of Parallel and Distributed Software Design*, Hudák Stefan, Kollár Ján(ed.) in.: Proceedings of the Sixth International Scientific Conference on Electronic Computers and Informatics, ECI 2004 September 22-24 Kosice-Herlány, Slovakia , ISBN.

[9] Porkoláb, Z., Zsók, V., *Teaching Multiparadigm Programming Based on Object-Oriented Programming*, in.: 10th Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts, TLOOC Workshop, ECOOP 2006, Nantes Nantes, France.

[10] Horváth Z., Kozsik T., Lövei L., *Szoftverrendszerek fejlesztésének oktatása projektfeladat keretén belül* ELTE 2009.

[11] Erlang Consulting. *2008 Obfuscated Erlang Programming Competition homepage.* `http://www.erlang-consulting.com/obfuscatederlang.html`.

[12] Barklund, J., Virding, R., Erlang Reference Manual, 1999. Available from `http://www.erlang.org/download/erl_spec47.ps.gz`. 2007.06.01.

[13] Plasmeijer, R. Eekelen, M., Functional Programming and Parallel Graph Rewriting, Addison-Wesley, 1993.

[14] Achten, P., Wierich, M., A Tutorial to the Clean Object I/O Library, University of Nijmegen, 2000. `http://www.cs.kun.nl/~clean`.

[15] Plasmeijer, R., Eekelen, M., *Clean Language Report v2.1*, `http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf`.

[16] `http://www.haskell.org/`.

[17] `http://msdn.microsoft.com/en-us/library/aa645596`.

Zoltán Hernyák
Roland Király
Eszterházy Károly College
Department of Information Technology
H-3300 Eger, Eszterházy tér 1.
e-mail: `{hz,serial}@aries.ektf.hu`