Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 285–294
doi: 10.14794/ICAI.10.2017.285

Cognitive tools in use

Péter Szlávi, Gábor Törley, László Zsakó

ELTE IK
szlavip@elte.hu
pezsgo@inf.elte.hu
zsako@caesar.elte.hu

Abstract

As a programmer is solving a problem, a number of conscious and unconscious cognitive operations are being performed. Problem-solving is a gradual and cyclic activity. As the mind is adjusting the task to its schemas formed by its previous experiences, the programmer gets closer and closer to understanding and defining the task. The primary cognitive operations the programmer uses to set up refining models are: linguistic abstraction, analogy, algorithmic abstraction, decomposition-superposition, conversion, intuition, and variation. In our previous articles [1, 2], we have described these cognitive tools, and now we will examine their operation.

Keywords: programming, didactics, systemic thinking, thinking toolkit, algorithmic abstraction, lexicality, creativity

MSC: 97D50, 97Q30, 97Q99

1. Introduction

While programmers are working hard to solve a programming task, consciously or unconsciously they use a wide variety of thinking methods. [1] As starting off from the task, they refine it several times to their own existing schemes based on experience, continually and circularly as well as more precisely reformulating the task. In other words, programming is a sequence of more and more refined (pattern-based) models, where you need to get to a stage where the vocabulary (i.e. the set of instruction patterns) of the programming language chosen serves as a basis for the model. (Figure 1)

The primary cognitive operations the programmer uses to set up refining models are:

- linguistic abstraction (see [1]),
- analogy (see [1]),

- algorithmic abstraction,
- decomposition-superposition,
- conversion,
- intuition,
- and variation ...

The programmer uses these operations many times and for manifold purposes. This is why we will also be mentioning a certain operation at more occasions.

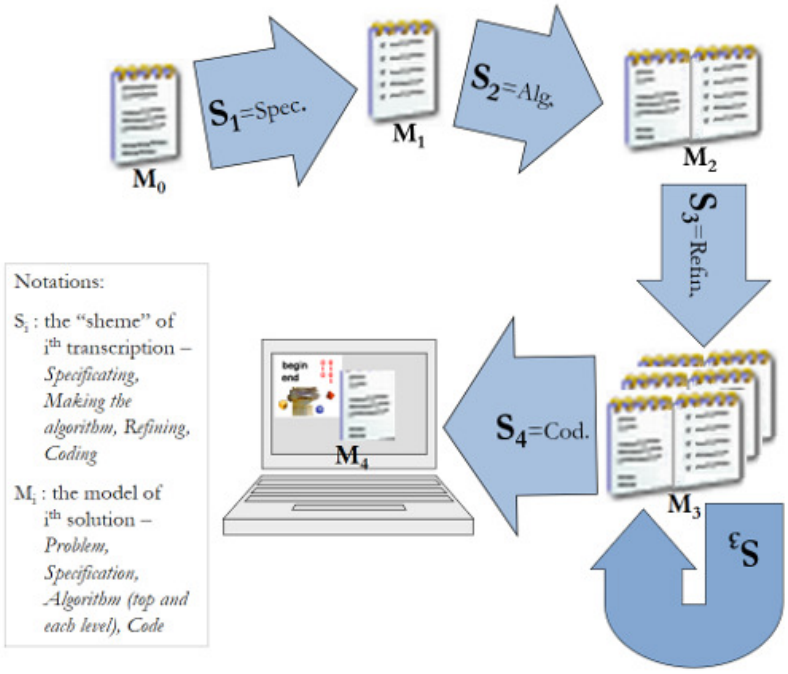

Figure 1: The abstract model of programming

As our first topic, we will introduce abstraction. We have decided not to distinguish linguistic abstraction from algorithmic, because in the thinking process they go pretty much hand in hand. Our chief concern in this section is to examine how the programmer utilizes abstraction in order to find an effective and safe solution to the programing problem.

As we move on to the next section, we will revisit a hot topic in education politics, namely, whether our education system should put more emphasis on lexicality, *instead* of creativity. Obviously, we are reflecting on this question from the perspective of programming, in light of the cognitive tools in focus.

2. Abstraction

Programming theses [4, 5, 6] are the foundations of high-level abstraction operations [1, 2, 3]. In this chapter we are presenting how we can create programming theses with the use of the cognitive operation of *abstraction* [2].

Generalization brings us to the notable classes of *search methods*. Let us start from the well-known algorithm of linear search. By introducing the following terms (Table 1) we will arrive to a fairly general search algorithm. [5, pp. 5–7] The search is performed in sequence X.

New notion	Explanation
$L \in H^*$ sequence of possibilities	properly selected sub-sequence of the original X ($L \subseteq X$)
$\|.\|$: $H^* \to \mathbb{N}$ length	the length of the sequence (number of elements)
σ: $H^* \to H^*$ sequence shortening	$\forall\ L \in H^*$: $\sigma(L)=L_1 \subset L$ ($\Rightarrow \|L\| > \|L_1\|$)
ϵ: $H^* \to \mathbb{N}$ element index selection	$\forall\ L \in H^*$: $\epsilon(L) \in [1..\|L\|]$

Table 1: The terms of generalized search

The algorithm of the *generalized search* thesis:

```
L:=X
While ||L||>0 and not T(X(ε(L))) do
    L:=σ(L)
End While
Exists:=||L||>0
If Exists then Index:=ε(L)
```

Let us match up the general notions with the "specific" notions of linear search.

$L:=(x_i{:}i \in [e..N])$ – all through, e : where we are=1..N+1 (LS1)
It is clear that L can be identified, even substituted, by the index of the first element:
$L:=L(e) \to e$ (LS1')
$\epsilon(L):=\epsilon(L(e)):=e$ – first element (LS2)
$\sigma(L):=\sigma(L(e)):=(x_i{:}i \in [e+1..N] \subseteq (x_i{:}i \in [e..N]))$ – the rest, shortened sequence (LS3)
(LS1)$\Rightarrow$ $\|L(e)\|:=N\text{-}e+1$ (LS4)
(LS4)$\Rightarrow$ $\|L(e)\|>0 \iff$
$N\text{-}e+1>0 \iff$
$N+1>e \wedge e \in \mathbb{N} \iff$
$N \geq e$ (LS5)

After "updating" the notions, let us take a look at the "basic" algorithm, which uses the general notions. We have added our comments to the table. (Table 2)

Generalized search	⇒	**Linear search**
L:=X	(LS1')	i:=1 [sequence starts with 1]
While ‖L‖>0	(LS5)	**While** i ≤ N
		[search can be performed still]
and not T(X(ϵ(L))) **do**	(LS2)	**and not** T(X(i))
		[ith is like that?] **do**
L:=σ(L(ϵ(L)))	(LS3)	i:=i+1 [sequence starts with next]
End While		**End While**
Exists:=‖L‖>0	(LS5)	Exists:=‖L‖>0 [is it?]
If Exists **then**		**If** Exists **then**
Index:=ϵ(L)	(LS1')	Index:=e

Table 2: Comparison of generalized and linear search

In the Table 3 we have collected the ϵ- and σ-functions of well-known searches.

	Definition	**Comment**
1	$\epsilon_F((x_i,...,x_j)):=i$	first element
2	$\epsilon_L((x_i,...,x_j)):=j$	last element
3	$\epsilon_M((x_i,...,x_j)):=i+j$ DIV 2	middle element
4	$\sigma_F((x_i,...,x_j)):=(x_{i+1},...,x_j)$	without first element
5	$\sigma_L((x_i,...,x_j)):=(x_i,...,x_{j-1})$	without last element
6'	$\sigma_{M<}((x_i,...,x_j)):=(x_i,...,x_m)$ where m=($i+j$ DIV 2)-1 It is clear that $\sigma_{M<}((x_i,...,x_j))\subseteq(x_i,...,x_j)$	elements preceding middle
6"	$\sigma_{M>}((x_i,...,x_j)):=(x_m,...,x_j)$ where m=($i+j$ DIV 2)+1 It is clear that $\sigma_{M>}((x_i,...,x_j))\subseteq(x_i,...,x_j)$	elements following middle

Table 3: Usual ϵ- and σ-function definitions

From the above algorithm of the general search, it is easy to deduct *linear search in ordered sequence*, *logarithmic search*, and even *backtracking*.

These are the notion correspondences for the algorithm of *linear search in ordered sequence*:

L:=(x_i:i ∈ [e..N]) – all through, e : where we are=1..N+1 (LSO1)
Here e signals the start of the sequence where we are performing the search. Note: sequence L can end even before we reach its N element. (If x>y is true.)
L can be identified, and even substituted with the index of the first element:
L:=L(e) → e (LSO1')

$\epsilon(\mathrm{L}):=\epsilon(\mathrm{L(e)}):=\epsilon_F(\mathrm{L(e)}):=$e – first element (LSO2)
$\sigma(\mathrm{L}):=\sigma(\mathrm{L(e)})$
$\sigma(\mathrm{L(e)}):=(\mathrm{x}_i{:}\mathrm{i} \in [\mathrm{e{+}1..N}])$. if $\mathrm{x}_e \leq \mathrm{y}$ (LSO3)
$\sigma(\mathrm{L(e)}):=(\mathrm{x}_i{:}\mathrm{i} \in [\mathrm{N{+}1..N}])$. if $\mathrm{x}_e > \mathrm{y}$ (LSO4)
(LSO3)&(LSO4)$\Rightarrow \sigma(\mathrm{L(e)}) \subseteq (\mathrm{x}_i{:}\mathrm{i} \in [\mathrm{e..N}])$ (LSO5)
$\|\mathrm{L}\|:=\|\mathrm{L(e)}\|$
(LSO1)$\Rightarrow \|\mathrm{L(e)}\|:=\mathrm{N\text{-}e{+}1}$ (LSO6)
(LSO6)$\Rightarrow \|\mathrm{L(e)}\|>0 \iff$
$\mathrm{N} \geq \mathrm{e}$ (LSO7)

This is the first version of the mechanically generated algorithm (already without references):

```
e:=1
While N≥e and X(e)≠y do
   Case
      When X(e)<y then e:=e+1
      When X(e)>y then e:=N+1
   End Case
End While
Exists:=N≥e
If Exists then Index:=e
```

We can make the following remarks:

1. According to LSO3, in the first node the following part ('**When** X(e)<y **then** e:=e+1') would be this:
   '**When** X(e)≥y **then** e:=e+1', which is semantically equivalent to the above due to the function condition.

2. We exit the cycle if 'N<e or X(e)=y' is true;

   N<e $\iff$
   X(e)>y – we find bigger ($\Leftarrow \sigma$ definition (LSO4))
   or
   N+1=e – we have reached the end of the sequence

3. The following is true about the *cycle condition*:

   N≥e and X(e)≠y $\iff$
   not (X(e)>y or N+1=e) and X(e)≠y $\iff$
   X(e)≤y and N+1>e and X(e)≠y $\iff$
   N+1>e and X(e)<y $\iff$
   N≥e and X(e)<y

4. After this change, the *conditional statement* became *useless* in the cycle, because the first condition is trivially true, and the second will never be true.

5. Like this, however, the reason of the exit is not signaled *unambiguously* by the 'N≥e' condition (since sequence L is not emptied): the exit path blocking successful searches and the exit path blocking meaningless searches are blurred now. Success, however, is unambiguously marked if 'N≥e and X(e)=y' conditions are true. Therefore, the modified algorithm is as follows:

```
e:=1
While N≥e and X(e)<y do
    e:=e+1
End While
Exists:=N≥e and X(e)=y
If Exists then Index:=e
```

6. The algorithm gets even more *simplified* with the separation of e. The trick is simply that we exit before the last element, focusing our scrutiny on this problem:

```
e:=1
While N>e and X(e)<y do
    e:=e+1
End While
Exists:=X(e)=y
If Exists then Index:=e
```

Note that it also models the operation of *program transformation* [7, p. 10: PT14] as a typical program making tool, whose essence is that we modify an algorithm (a program) with justifiably proper transformation steps, usually for the purpose of reducing its complexity.

We can reach the well-known algorithm of *logarithmic search* with the same logic. In the following, we will only present its start and its end.

These are the notion correspondences for the algorithm of *logarithmic search*:

$L:=(x_i:i\in[e..v])$ – start and finish; at start:(1..N) (LgS1)

L can be identified, even substituted with the index of its first and last element:

$L:=L(e,v) \rightarrow (e,v)$ (LgS1')

$\epsilon(L):=\epsilon(L(e,v)):=\epsilon_M(L(e,v)):=(e+v)$ DIV 2 – middle element (LgS2)

$\sigma(L):=\sigma(L(e,v))$

$\sigma(L(e,v)):=\sigma_{M<}(L(e,v))$, if $x_{\epsilon(L(e,v))}>y$ (LgS3)

$\sigma(L(e,v)):=\sigma_{M>}(L(e,v))$, if $x_{\epsilon(L(e,v))}<y$ (LgS4)

$\|L\|:=\|L(e,v)\|$

(LgS1)$\Rightarrow \|L(e,v)\|:=v-e+1$ (LgS5)

(LgS5)$\Rightarrow \|L(e,v)\|>0 \iff$

$\|L(e,v)\|>0 \iff$

$v-e+1>0 \wedge e,v\in \mathbb{N} \iff$

$v\geq e$ (LgS6)

The algorithm after the direct generation is as follows:

```
(e,v):=(1,N)
While v≥e and X((e+v) DIV 2)≠y do
   Case
      When X((e+v) DIV 2)>y then (e,v):=(e,((e+v) DIV 2)-1)
      When X((e+v) DIV 2)<y then (e,v):=(((e+v) DIV 2)+1,v)
   End Case
End While
Exists:=v≥e
If Exists then Index:=(e+v) DIV 2
```

For the sake of simplicity, we assign k to the index of element (=(e+v) DIV 2) of ϵ(L(e,v)), but this separation entails distinct calculation as well. On the other hand, the 'X(k)≠y' condition simplifies the cycle to being two-way. This means we are left with the "usual" algorithm:

```
(e,v):=(1,N); k:=(e+v) DIV 2
While v≥e and X(k)≠y do
   If X(k)>y then v:=k-1
             else  e:=k+1
   k:=(e+v) DIV 2
End While
Exists:=v≥e
If Exists then Index:=k
```

Backtracking can also be deducted from the above algorithm of general search, but it is somewhat more complicated due to the somewhat more complicated definition of *search domain* in the backtracking logic. For details check [5, pp. 8–9].

Note that you can read more in depth about the above aspect of abstraction on [6, pp. 64–65], where we are analyzing how to build up the programming theses (like counting, maximum pick, multiple item selection, etc.), which are operated by the backtracking logic, from the abstract notions listed in Table 1. This points to another useful consequence of the abstract mindset described in this chapter, the generalization of search. Obviously, there exist other ways to generalize, for example based on *data structure variation* (using matrices [Table 5], lists, sequential input/output files, sets or graphs, instead of arrays)

3. Lexicality vs. creativity

It must be noted that we rely on a great deal of lexical knowledge while programming. During this process, we use languages with small lexicons but great variability, such as specification languages, algorithmic languages, or programming languages. Systematic programming regulates coding as well, so in addition to lexical knowledge, we are dealing with coding rules too. Besides the basic level of programming, programming theses [5, 6, 8] can also be considered as lexical knowledge, which are topped up by program transformations [7, 8] on the advanced level

Definition	Linear search in the matrix

L:=($x_{e,j}$:j∈[f..N]) ∪
($x_{i,j}$:i∈[e+1..N],j∈[1..M])
L:=L(e,f) → (e,f)
ϵ(L):=ϵ(L(e,f)):=(e,f)
σ(L):=σ(L(e,f))
σ(($x_{e,f}$,...,$x_{N,M}$)):=
($x_{e,f+1}$,...,$x_{N,M}$), if f<M
σ(($x_{e,f}$,...,$x_{N,M}$)):=
($x_{e+1,1}$,...,$x_{N,M}$), if f=M
‖L(e,f)‖:=(M-e+1)*M-f
	(e,f):=(1,1)
	While e≤N **and not** T(X(e,f)) **do**
	if f<M **then**
	f:=f+1
	else
	e:=e+1; f:=1
	End If
	End While
	Exists:=e≤N
	If Exists **then** Index:=(e,f)

Table 5: Definitions and algorithm of linear search for matrix with N×M dimension

of programming. The question arises then: are we not contributing to the negative state of skills development in students, as revealed in the PISA reports [9, 10], by forcing programming? "Too much lexicality; where is creativity??"

It is evident that the goal of coding rules is exactly to make it easier to learn a given programming language. They concentrate the knowledge, thus reducing the time necessary to start the coding process. But what about the rest of the lexical knowledge?

Let us quote from psychologist-philosopher William James [11, p. 122]:

"*The great thing, then, in all education, is to make our nervous system our ally instead of our enemy. It is to fund and capitalize our acquisitions, and live at ease upon the interest of the fund. For this we must make automatic and habitual, as early as possible, as many useful actions as we can, and guard against the growing into ways that are likely to be disadvantageous to us, as we should guard against the plague. The more of the details of our daily life we can hand over to the effortless custody of automatism, the more our higher powers of mind will be set free for their own proper work.*"

If we translate James's thoughts to programming, we can conclude that by making students "memorize" the lexical knowledge connected to programming we are giving them tools with which they can start to focus their **creativity** on **solving real problems**, not wasting their energy on tiny algorithmic details. In this way, creativity can be utilized on a higher level, on the level of problem-solving. Of course, if we rigidly stick to our thinking patterns, that can pose a problem. Elemér Lábos [12, p. 61] describes this phenomenon in the following way:

"*Attitude is an especially interesting internal mechanism, as it points out two processes that play an important role in thinking. When we think, our mind activates programs that are there ready to solve repeated problems, which accelerates problem-solving, yet at the same time it blocks the activation of other programs.*"

4. Summary

We have examined two main issues in this article: 1) how abstraction operates as a cognitive tool, and 2) how lexical knowledge of linguistic nature can serve to help problem-solving.

We can state that both of them make 1) problem-solving thinking more efficient, and 2) the final solution more complex. Nevertheless, it is clear that the application of these tools can pose challenges too: it is fair to admit that they require more mental effort, schematic thinking is not enough here, and certain lexical knowledge is also necessary, along with their routine-like application and some self-control to keep ourselves from the attractive yet misleading solutions that keep popping up.

References

[1] Szlávi, P., Zsakó, L., Törley, G., The Thinking Toolkit of Programming, *Proceedings of XXIX. DidMatTech 2016, "New methods and technologies in education and practice" Conference* (2016), 55–62.
Available at `https://edit.elte.hu/xmlui/handle/10831/32278` (last retrieved: 11/06/2016)

[2] Szlávi, P., Zsakó, L., A programozás gondolkodási eszköztára – Algoritmikus absztrakció, dekompozíció-szuperpozíció, *InfoDidact 2016, Informatika Szakmódszertani Konferencia*, (2016),
Available at `http://people.inf.elte.hu/szlavi/InfoDidact16/Manuscripts/SzPZsL.pdf` (last retrieved: 12/03/2016)

[3] Szlávi, P., A programkészítés didaktikája, *PhD dissertation*, (2004),
Available at `http://www.inf.elte.hu/karunkrol/szolgaltatasok/konyvtar/Lists/Doktori%20disszertcik%20adatbzisa/Attachments/32/Szlavi_Peter_Ertekezes.pdf` (last retrieved: 12/03/2016)

[4] Szlávi, P., Programok, programspecifikációk, *Informatika a Felsőoktatásban'99*, (1999), 576–582.
Available at `http://people.inf.elte.hu/szlavi/ProgModsz/Progspec.pdf` (last retrieved: 11/06/2016)

[5] Szlávi, P., Programozási tételek – Összefoglaló, *Manuscript*, (2001)
Available at `http://people.inf.elte.hu/szlavi/ProgModsz/Prtetel.pdf` (last retrieved: 11/06/2016)

[6] Szlávi, P., Zsakó, L., Módszeres programozás – Programozási tételek, *TTK Informatikai Tanszékcsoport*, (2004),

[7] Szlávi, P., Programozási tételek egymásra építése – Programtranszformációk, *Manuscript*, (2000)
Available at `http://people.inf.elte.hu/szlavi/ProgModsz/Progtran.pdf` (last retrieved: 01/06/2017)

[8] Harangozó, É., Szlávi, P., Zsakó, L., Joining Programming Theorems a Practical Approach to Program Building, *Annales Universitatis Scientiarum Budapestinensis. Sectio Computatorica*, (1998), 155–172.

Available at http://ac.inf.elte.hu/Vol_017_1998/155.pdf (last retrieved: 11/06/2016)

[9] Csapó, B. et al., Az iskolai teljesítmények alakulása Magyarországon nemzetközi összehasonlításban, *Kolosi Tamás és Tóth István György (edit.): Társadalmi Riport 2014. TÁRKI*, (2014), 110–136.
Available at http://www.tarki.hu/adatbank-h/kutjel/pdf/b327.pdf (last retrieved: 01/06/2017)

[10] Palkovics, A lexikális tudás- és a kompetencia fejlesztése egyaránt fontos, *Magyar Hírlap.hu*, (2016) Available at http://magyarhirlap.hu/cikk/73625/Palkovics_A_lexikalis_tudas_es_a_kompetencia_fejlesztese_egyarant_fontos (last retrieved: 01/06/2017)

[11] James, W., The Principles of Psychology, Vol. 1 *Henry Holt and Co., New York*, (1910) Available at https://ia800203.us.archive.org/12/items/theprinciplesofp01jameuoft/theprinciplesofp01jameuoft.pdf (last retrieved: 01/13/2017)

[12] Lábos, E., Természetes és mesterséges értelem, *Magvető Kiadó*, (1979)