Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1533–1536

Teaching Programming through Problem Solving: The Role of the Programming Language

Nikolaos S. Papaspyrou
Email: nickie@softlab.ntua.gr

Stathis Zachos
Email: zachos@cs.ntua.gr

School of Electrical and Computer Engineering
National Technical University of Athens
Polytechnioupoli, 15780 Zografou, Athens, Greece

***Abstract*—In this short paper, we advocate the importance of problem solving for teaching "Introduction to Programming", instead of merely teaching the syntax and semantics of a programming language. We focus on the role of the programming language used for an introductory course. For this purpose we propose CAL, a C-like algorithmic language, which is essentially a well-defined and behaved subset of C with a small number of modest, "educational" extensions. We present the design rationale for CAL, its main features, syntax and illustrative examples.**

I. Introduction

IN THIS short paper, we present our experiences with teaching programming through *problem solving* in the School of Electrical and Computer Engineering of the National Technical University of Athens. We focus on the role of the programming language for this purpose and describe the approach that we have taken. Let us begin with two observations:

1) In some students' minds, algorithmic programming is strangely enough an intellectual process that is not connected to everyday problem solving.
2) Students often have no sense of what is good and what is bad in programming, even after taking a number of courses on the design of algorithms and complexity.

We believe that these are both due to the way we teach students how to program. Starting from secondary school, students often begin by learning a Pascal-like programming language. They learn to use variables, assignments, control-flow statements, arrays. They do not learn, however, because we do not teach them early enough, *what these should be used for!* Young children realize early that they need to solve problems; they are hungry and they want their parents to feed them, they want to play with that shiny car in the toy shop's window, etc. Later on, they learn to speak and use the *language* to communicate their needs. Children learn to speak after they know what they want to say! Why do we teach programming languages to students before they know what to use them for?

The main goal of our proposal is a quick introduction to programming for absolute beginners. Such an introduction would be useful for teenagers in high-school, who may not continue to be programming specialists but who want to support their literacy in mathematics by hands-on attractive algorithmically solvable problems. It would also be useful to first-year university students who have (somehow) escaped a proper exposure to programming in high school (and this is the majority of our students). Thus, this goal translates to:

- a quick educational introduction to self-evident programming concepts and tools, without cryptic, hardware-dependent and special purpose structures; but also
- fluency in a programming language which can be easily extended and/or modified to a language that is currently useful in practice, without a total rethinking of the basic algorithmic techniques.

Our historic prototype for a self-evident educational programming language is of course Pascal [10], whereas programming languages that are currently useful in practice are of course C [7], [5], C++ [9] and Java [2].

II. The Role of the Language

Since the 1950s, scores of different programming languages have been designed and implemented and many more are yet to come. Those with a relative experience in the field will agree that there is no such thing as "the best programming language" and this is what we need to explain to our students early on. Some languages are better than others *for some specific purpose* and indeed: (a) some specific languages are almost exclusively used for some specific purposes, and (b) for some specific purposes people use almost exclusively some specific languages.

For example, C [7], [5] is a very good language for systems programming. It is a low-level language, offering programmers the opportunity to directly interact with the hardware, but not the best language for numerical and scientific computing today. We believe that C is an inappropriate language for teaching "Introduction to Programming". Some of its characteristics are so low-level that tend to focus on the hardware, instead of on the algorithms. When you start learning how to program, you don't need twelve different types for integer numbers (including characters and Boolean values) and three more for real (floating-point) numbers. You don't need a `for` statement so powerful that you can use it to implement a binary search algorithm in just one line. You don't need to struggle to understand the meaning of `x = x++;` (most people, including some who teach C, think that it does something although opinions vary when it comes to what exactly this is; the truth is that this statement is illegal, or causes "undefined behaviour"

978-1-4673-4471-5/$25.00

as the ANSI C standard puts it, because the value of variable `x` changes twice between two successive sequence points).

Pascal [10], [6] is arguably one of the best programming languages for teaching purposes. It is a concise, general purpose language which supports a systematic, structured and algorithmic approach to problem solving. Programs in Pascal are usually easy to read and understand with a clear structure that favours stepwise refinement. The language helps programmers to avoid programming mistakes and to be able to verify the correctness of their programs. On the other hand, Pascal is very little used today by software practitioners.

Java and other object-oriented languages are also poor candidates for teaching "Introduction to Programming". If the focus is on problem solving and algorithmic thinking, such languages add an unbearable level of noise. Using Java, the only logical approach is to teach programming in a purely object-oriented fashion and this necessarily takes the focus away from problem solving, although OOP might be a good choice for a second (e.g., data structures) course.

There is a trend towards Python, in the last few years. Python is a relatively good candidate; its syntax is concise and enforces proper indentation, it supports imperative and object-oriented programming equally well, and it is widely used in the software industry. The drawbacks for Python are: (a) it is dynamically typed and requires very few declarations; this is bad if you want the students to detect programming errors early and to learn to program in a disciplined way; (b) it is so high-level that students do not develop an intuition about how data are represented and how operations are implemented; this is bad if you want the students to understand the connection between the programming language and the underlying hardware; and (c) its data structures are so high-level that algorithmic complexity issues are obscured by the way data structures are implemented; e.g., in Python there are no arrays, but there are lists and dictionaries; however, in a data structure course, all three would need to be covered and with three different implementations, requiring $O(1)$, $O(n)$ and $O(n \log n)$ time for accessing an element, respectively.

The second drawback (b) is also true for functional languages, like Scheme, ML or Haskell, which are also very good from an educational point of view and are indeed used for teaching "Introduction to Programming" in several Computer Science departments [8], [3], [1], [4].

All this said, we decided to design a new educational programming language for an introductory course in which emphasis is on problem solving. However, this educational language will naturally evolve before the students' eyes to a full-scale programming language, useful later on. In the next sections we describe CAL, a *C-like algorithmic language*, starting from the design choices that we had to make, proceeding with the syntax, the main characteristics of the language and concluding with a few examples.[1]

[1]An implementation of CAL, based on GCC and using macros, is available from https://github.com/softlab-ntua/pazcal.

III. The Design of CAL

Disregarding some drawbacks, C is an adequate choice for teaching "Introduction to Programming", with emphasis on problem solving and algorithmic thinking. Its core is a quite simple algorithmic language, easy to explain and use. Moreover, it is a useful language to know, heavily used in practice, either directly or indirectly, through a line of descendants that share a large part of its syntax and semantics (C++, Java, C#, etc.). A list of drawbacks:

- Its syntax and semantics is often cryptic and obfuscated; e.g., allowing side-effects anywhere inside expressions.
- The use of "declarators" (as in `int*(f[3])(int);`) is counter-intuitive and hard to explain.
- Non trivial library functions (e.g., `printf` and `scanf`) are required for beginners to write programs that input and output data. The corresponding header files must be `#include`d. Pointers are required for `scanf`.
- Before the simplest program is written, students must see `int main()` and `return 0;` unless of course we want to teach them to be sloppy from the first lecture...
- The type system allows programmers to deliberately misuse data and to neglect declaring function prototypes. Both are bad from an educational point of view.

We therefore base CAL on an appropriate "educational subset" of C, which we extend with a number of macros, library functions and one extra feature (call by reference) to suit the needs of our introductory course. The result is a language reminiscent of Pascal but with C notation. All extensions are written with *uppercase letters* (e.g., **WRITE**), so that students immediately know if something that they have learnt exists in C or is one of our educational extensions. The main characteristics of CAL, whose complete syntax is defined in figure 1, are the following:

- A program is organized as a set of modules, each consisting of constant and type definitions, variable definitions, routine declarations, routine definitions and (optionally) the body of the main program, which must only be present in one module. The visibility of module definitions is controlled with **PRIVATE** and **extern**.
- There are *functions* and *procedures*, defined with **FUNC** and **PROC** respectively; the misleading type **void** is not used. The main program begins with the special keyword **PROGRAM**.
- The type system is simplified. There are types for Boolean values (**bool**, as in C99, with constants **true** and **false**), integers (**int**), characters (**char**) and real numbers (**REAL**). There are also enumerations, structures and unions, but these must be defined and given a name before they can be used. Arrays and pointers complete the picture of types. However, the syntax for declarators is very simplified in comparison with C; type synonyms (**typedef**) can be used for defining, e.g., double pointers, arrays or pointers, pointers to arrays, etc.
- Operators **NOT**, **AND**, **OR** and **MOD** are synonyms of C's (not so intuitive) standard operators `!`, `&&`, `||` and `%`.

⟨module⟩ ::= (⟨const_def⟩ | ⟨type_def⟩)* (⟨declaration⟩)* (⟨definition⟩)* [⟨program⟩]
⟨declaration⟩ ::= ["**PRIVATE**" | "**extern**"] (⟨var_def⟩ | ⟨routine_decl⟩)
⟨definition⟩ ::= ["**PRIVATE**" | "**extern**"] ⟨routine_def⟩
⟨const_def⟩ ::= "**const**" ⟨type⟩ ⟨declarator⟩ "**=**" ⟨initializer⟩ ("**,**" ⟨declarator⟩ "**=**" ⟨initializer⟩)* "**;**"
⟨type_def⟩ ::= "**typedef**" ⟨type⟩ ⟨declarator⟩ ("**,**" ⟨declarator⟩)* "**;**" | ⟨enum_def⟩ | ⟨struct_def⟩ | ⟨union_def⟩
⟨enum_def⟩ ::= "**enum**" ⟨id⟩ "**{**" ⟨id⟩ ("**,**" ⟨id⟩)* "**}**" "**;**"
⟨struct_def⟩ ::= "**struct**" ⟨id⟩ "**{**" (⟨type⟩ ⟨declarator⟩ ("**,**" ⟨declarator⟩)* "**;**")* "**}**" "**;**"
⟨union_def⟩ ::= "**union**" ⟨id⟩ "**{**" (⟨type⟩ ⟨declarator⟩ ("**,**" ⟨declarator⟩)* "**;**")* "**}**" "**;**"
⟨var_def⟩ ::= ⟨type⟩ ⟨declarator⟩ ["**=**" ⟨initializer⟩] ("**,**" ⟨declarator⟩ ["**=**" ⟨initializer⟩])* "**;**"
⟨routine_decl⟩ ::= ⟨routine_header⟩ "**;**"
⟨routine_def⟩ ::= ⟨routine_header⟩ ⟨block⟩
⟨routine_header⟩ ::= ("**PROC**" | "**FUNC**" ⟨type⟩) ⟨id⟩ "**(**" [⟨type⟩ ⟨formal⟩ ("**,**" ⟨type⟩ ⟨formal⟩)*] "**)**"
⟨formal⟩ ::= ⟨id⟩ ["**[**" "**]**"] ("**[**" ⟨expr⟩ "**]**")* | "*****" ⟨id⟩ | "**&**" ⟨id⟩
⟨type⟩ ::= "**int**" | "**bool**" | "**char**" | "**REAL**" | "**enum**" ⟨id⟩ | "**struct**" ⟨id⟩ | "**union**" ⟨id⟩ | ⟨id⟩
⟨declarator⟩ ::= ⟨id⟩ ("**[**" ⟨expr⟩ "**]**")* | "*****" ⟨id⟩
⟨initializer⟩ ::= ⟨expr⟩ | "**{**" ⟨initializer⟩ ("**,**" ⟨initializer⟩)* "**}**"
⟨program⟩ ::= "**PROGRAM**" ⟨id⟩ "**(**" "**)**" ⟨block⟩
⟨block⟩ ::= "**{**" (⟨local_def⟩ | ⟨stmt⟩)* "**}**"
⟨local_def⟩ ::= ⟨const_def⟩ | ⟨var_def⟩
⟨stmt⟩ ::= "**;**" | ⟨l_value⟩ ⟨assign⟩ ⟨expr⟩ "**;**" | ⟨l_value⟩ ("**++**" | "**--**") "**;**" | ⟨write⟩ "**(**" [⟨format⟩ ("**,**" ⟨format⟩)*] "**)**" "**;**"
| "**FOR**" "**(**" ⟨id⟩ "**,**" ⟨range⟩ "**)**" ⟨stmt⟩ | "**while**" "**(**" ⟨expr⟩ "**)**" ⟨stmt⟩ | "**do**" ⟨stmt⟩ "**while**" "**(**" ⟨expr⟩ "**)**" "**;**"
| "**if**" "**(**" ⟨expr⟩ "**)**" ⟨stmt⟩ ["**else**" ⟨stmt⟩] | "**break**" "**;**" | "**continue**" "**;**" | "**return**" [⟨expr⟩] "**;**"
| ⟨block⟩ | ⟨call⟩ "**;**" | "**switch**" "**(**" ⟨expr⟩ "**)**" "**{**" (("**case**" ⟨expr⟩ "**:**")+ ⟨clause⟩)* ["**default**" "**:**" ⟨clause⟩] "**}**"
⟨assign⟩ ::= "**=**" | "**+=**" | "**-=**" | "***=**" | "**/=**" | "**%=**"
⟨range⟩ ::= ⟨expr⟩ ("**TO**" | "**DOWNTO**") ⟨expr⟩ ["**STEP**" ⟨expr⟩]
⟨clause⟩ ::= (⟨stmt⟩)* ("**break**" "**;**" | "**NEXT**" "**;**")
⟨write⟩ ::= "**WRITE**" | "**WRITELN**" | "**WRITESP**" | "**WRITESPLN**"
⟨format⟩ ::= ⟨expr⟩ | "**FORM**" "**(**" ⟨expr⟩ "**,**" ⟨expr⟩ ["**,**" ⟨expr⟩] "**)**"
⟨expr⟩ ::= ⟨int-const⟩ | ⟨float-const⟩ | ⟨char-const⟩ | ⟨string-literal⟩ | "**true**" | "**false**" | "**(**" ⟨expr⟩ "**)**" | ⟨l_value⟩ | ⟨call⟩
| ⟨unop⟩ ⟨expr⟩ | ⟨expr⟩ ⟨binop⟩ ⟨expr⟩ | "**(**" ⟨type⟩ "**)**" ⟨expr⟩ | "**NULL**" | "**NEW**" "**(**" ⟨type⟩ ["**,**" ⟨expr⟩] "**)**"
⟨l_value⟩ ::= ⟨id⟩ | ⟨expr⟩ "**[**" ⟨expr⟩ "**]**" | "*****" ⟨expr⟩ | ⟨expr⟩ "**.**" ⟨id⟩ | ⟨expr⟩ "**->**" ⟨id⟩
⟨unop⟩ ::= "**+**" | "**-**" | "**NOT**" | "**!**"
⟨binop⟩ ::= "**+**" | "**-**" | "*****" | "**/**" | "**%**" | "**MOD**" | "**==**" | "**!=**" | "**<**" | "**>**" | "**<=**" | "**>=**" | "**&&**" | "**AND**" | "**||**" | "**OR**"
⟨call⟩ ::= ⟨id⟩ "**(**" [⟨expr⟩ ("**,**" ⟨expr⟩)*] "**)**"

Fig. 1. A context-free grammar defining the syntax of CAL in EBNF. Operator precedence and associativity are the same as in C.

- Assignment (simple or composite) is a statement. (Alas, for compatibility purposes we have to give up the assignment operator `:=` and accept the commonly used `=`, which we would prefer not to confuse with the equality operator known from mathematics.) There is just one type of increment and decrement operators (postfix, e.g., `x++`), used again as statements. Therefore, expression evaluation cannot contain direct side-effects. Also, we omit bitwise operators and conditional expressions (`?:`).

- We omit the **`for`** statement, which is too general for our purposes, and replace it with a **`FOR`** statement following the style of Pascal, mentioning the control variable and a (precomputed) range of values that the control variable will take. Using **`FOR`**, the maximum number of iterations is always finite and known before the loop starts executing; **`break`** and **`continue`** can be used to exit the loop and proceed with the next iteration.

- The **`switch`** statement is sanitized; **`case`** labels cannot appear everywhere. Furthermore, clauses are required to end either with a **`break`**, or with the new keyword **`NEXT`** in order to explicitly proceed to the next clause.

- Four kinds of **`WRITE`** statements are used for the output of data, allowing any number of arguments of any type, with a number of formatting options that are useful for an introductory course. Library functions `READ_INT`, `READ_REAL` and `getchar` are used to input data.

- The use of pointers has also been sanitized. The connection between pointers and arrays is still present, but it only allows the use of a pointer as an array; no pointer arithmetic is allowed and there is no "address of" operator (`&`). Pointers are used for dynamic memory allocation; **`NEW`** and **`DELETE`** help students for this purpose (in the spirit of C++, instead of `malloc` and `free` in C).

- Call by reference is allowed, using the same notation that

C++ uses for references.

IV. Introduction to Programming using CAL

Our course is based on the simplicity philosophy of the great teachers of the 1960s: Dijkstra, Hoare, and Wirth. We only give some highlights for lack of space.

- As early as in the second week, students are able to write simple programs that input, process and output data.

```
PROGRAM area_of_circle ()
{ WRITE("Give the radius: ");
  REAL r = READ_REAL();
  REAL a = 3.1415926 * r * r;
  WRITELN("The area is: ", a);
}
```

- Control flow and combinatorial calculations.

```
PROGRAM primes ()
{ int p;
  WRITELN(2);
  FOR (p, 3 TO 1000 STEP 2)
  { int t = 3;
    while (p MOD t != 0) t += 2;
    if (p == t) WRITELN(p);
  }
}
```

- Structured programming: modules, functions and procedures, parameter passing, stepwise refinement.
- Recursion. Euclid's algorithm for the greatest common divisor is one of the examples that we use:

```
FUNC int gcd (int i, int j)
{ if (i==0 OR j==0) return i+j;
  else if (i > j)   return gcd(i MOD j, j);
       else         return gcd(i, j MOD i);
}
```

- Call by reference: e.g., in `swap`, useful for sorting.

```
PROC swap (int &x, int &y)
{ int t = x; x = y; y = t; }
```

- Arrays: merge sort and quick sort, which are also examples of recursion.

```
PROC merge (int a[], int fst, int mid, int lst);

PROC mergesort (int a[], int first, int last)
{ if (first >= last) return;
  int mid = (first + last) / 2;
  mergesort(a, first, mid);
  mergesort(a, mid+1, last);
  merge(a, first, mid, last);
}
```

- Dynamic data structures, such as linked lists and trees, e.g., in-situ reversal of a simply linked list.

```
struct node { int data; struct node *next; };
typedef struct node *list;

PROC reverse (list &l)
{ list q = NULL;
  while (l != NULL)
  { list p = l;
    l = p->next; p->next = q; q = p;
  }
  l = q;
}
```

When grading, we reward the design of efficient algorithms for solving problems such as the following:

MAXSUM: Read a sequence of n integer numbers (positive, zero, or negative) and output the largest value of the sum of (arbitrarily many) consecutive numbers in the sequence.

We expect students who correctly solve this problem to come up with one of three general types of solutions, or variations thereof. The first, is the obvious $O(n^3)$ algorithm: for all possible starts (`i`) and ends (`j`) of subsequences, calculate the sum and find the largest. The second is the slightly less obvious $O(n^2)$ algorithm that, for every given start (`i`) avoids recomputing the sum of a subsequence from scratch but reuses the sums of smaller subsequences. Finally, the third is a linear algorithm — $O(n)$ time and requiring $O(1)$ memory — which works in a greedy fashion.

```
PROGRAM maxsum_On ()
{ int n = READ_INT();
  int i, sofar = 0, best = 0;
  FOR (i, 0 TO n-1)
  { int x = READ_INT();
    sofar += x;
    if (sofar < 0)          sofar = 0;
    else if (sofar > best) best = sofar;
  }
  WRITELN(best);
}
```

V. Concluding Remarks

We have presented CAL, a C-like algorithmic language that we advocate for teaching "Introduction to Programming" focusing on problem solving. We discussed the design of CAL, which is essentially a controlled subset of C with some appropriate extensions. We would like to see CAL, or appropriate subsets of it, as a vehicle to teach computer programming to high-school students, making a bridge between mathematics and computer science in secondary education.

References

[1] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: An Introduction to Computing and Programming*. MIT Press, 2001.

[2] J. Gosling, B. Joy, G. L. S. Jr., G. Bracha, and A. Buckley, *The Java Language Specification*, java se 7 ed. Addison-Wesley, 2013.

[3] P. Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[4] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2007.

[5] *ISO/IEC 9899:2011 Standard, Information technology – Programming languages – C*, International Organization for Standardization, 2011.

[6] K. Jensen and N. Wirth, *Pascal user manual and report — ISO Pascal standard, 4th Edition*. Springer, 1991.

[7] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988.

[8] L. C. Paulson, *ML for the Working Programmer*, 2nd ed. Cambridge University Press, 1996.

[9] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley, 1997.

[10] N. Wirth, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35–63, 1971.