Picky: A New Introductory Programming Language

Francisco J. Ballesteros
Universidad Rey Juan Carlos
C/ Camino del Molino SN
E29843, Fuenlabrada, Madrid, Spain
nemo@lsub.org

Gorka Guardiola Múzquiz
Universidad Rey Juan Carlos
C/ Camino del Molino SN
E29843, Fuenlabrada, Madrid, Spain
paurea@lsub.org

Enrique Soriano Salvador
Universidad Rey Juan Carlos
C/ Camino del Molino SN
E29843, Fuenlabrada, Madrid, Spain
esoriano@lsub.org

ABSTRACT

In the authors' experience the languages available for teaching introductory computer programming courses are lacking. In practice, they violate some of the fundamentals taught in an introductory course. This is often the case, for example, with I/O. Picky is a new open source programming language created specifically for education that enables the students to program according to the principles laid down in class. It solves a number of issues the authors had to face while teaching introductory courses for several years in other languages. The language is small, simple and very strict regarding what is a legal program. It has a terse syntax and it is strongly typed and very restrictive. Both the compiler and the runtime include extra checks to provide safety features. The compiler generates byte-code for compatibility and the programming tools are freely available for Linux, MacOSX, Plan 9 from Bell Labs and Windows. This paper describes the language and discusses the motivation to implement it and its main educational features.

Categories and Subject Descriptors

D.3.0 [**Programming Languages**]: General
; D.3.3 [**Programming Languages**]: Language Constructs and Features
; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Programming Languages, CS1

Keywords

Programming Languages, CS1

1. INTRODUCTION

The authors are in charge of teaching an introductory computer science course (CS1 from now on). The curriculum is focused on imperative, statically typed procedural programming. Nevertheless, it is not the usual imperative programming curriculum. It strongly emphasizes the *top-down* approach and the definition of subprograms. Proof of this is that the course starts as a functional programming course. The students learn how to build expressions and functions before learning how to declare variables and build sequences of statements. However, only one (imperative) programming language is used for the whole course. Our approach is similar in spirit to [8], but with a different implementation. Where Decker et al focus on object organization, we focus on the strategy for attacking the problem by breaking it into subproblems.

The course follows a twofold pedagogy methodology. First, at every point the student is required to write code to test her understanding of the matter at hand. Second, every line of code the student writes must be comprehensible at that point of the course. Of course, the second part needs to be relaxed somewhat at the start of the course, but it is an important principle we adhere to, whenever possible.

After teaching this course several years using Ada as the main language, the authors decided to look for an alternative for several reasons. Ada, despite being a Pascal descendant, is multiparadigm. Moreover, its syntax is too verbose, and it has other issues that are discussed next.

Selecting a new language for CS1 is a complex and delicate task. The related bibliography is extensive [18] and there are many open discussions about the different approaches, for instance [14, 22, 21, 15, 5]. Although other authors defend the use of object oriented languages for introductory courses (see for example [17]), there is no consensus about which approach is better (*objects early* vs *procedures early*) [22, 21, 15, 5]. In the authors' experience, object oriented languages are too complex to be used as a first language. As other authors state [12], the student should be instructed before delving into an understanding of object oriented programming concepts, which are more abstract (inheritance, delegation, polymorphism, etc.) than other basic prerequisites (variables, parameters). Object oriented languages may be popular, but they are not simple enough to be understandable for a primer and look like magic to most novice students.

After some research, the authors were not satisfied with the existing alternatives to replace Ada. Although there are

many programming languages available, and some of them are specifically created for education, none satisfied all our needs, described in section 2. In the end, the authors decided to design and implement Picky and write a text book (in Spanish) for the course [3]. Picky is a new imperative programming language that meets all these requirements. This paper presents the main features of the language for teaching the CS1 course and the experience after using it for nine CS1 courses with more than six hundred students.

2. REQUIREMENTS

2.1 High level

It is widely accepted [16] that low level languages, such as C, are not suitable for CS1 courses. Even the defenders of such languages acknowledge their shortcomings [20].

Thus, the candidate language to replace Ada for this purpose must completely abstract the details of the machine and the underlying operating system.

2.2 Single-paradigm

Some prestigious institutions, for example CalTech and MIT, use multi-paradigm languages in introductory courses [14]. As stated before, some of the issues that the authors found while teaching CS1 in Ada are related to its multi-paradigm features.

Although multi-paradigm languages can be suitable for long courses that start with imperative programming and then continue with object oriented programming, they are not suitable for single-paradigm courses (i.e. imperative programming). It is confusing for students to consult bibliography that mixes the paradigms or references focused on a paradigm that is out of the scope of the course.

Also, as part of the learning process, the student, by mistake, may write programs that wander off the subset taught in class. When the language has many heterogeneous constructions, like Ada, it is highly probable that the student may come across one of them by mistake. Another issue is the compiler returning an error related to one of these off-course constructions. Frustration and confusion ensues.

Using a pure object oriented language (e.g. Java) to teach imperative programming, like some institutions do, is even worse. For a significant part of the course, the student gets used to writing code which is incomprehensible at that point in time (public, static, class, etc.). This violates one of the cores of the twofold approach detailed above.

2.3 Restrictive

The candidate language must provide strong typing and range checking. These features are very convenient when learning how to program for the first time. With them, the compiler and the runtime act as a safety net which prevents the students from wandering off too much. This is another reason for not using languages such as C, where the plasticity of the language makes it easy to write obscure code. In addition, it is desirable to use a language that includes extra restrictions. For example, global variables are very harmful in an introductory course and it is convenient to use a language that forbids them. This forces the student to get into the habit of structuring the code properly.

Some kinds of syntactic sugar and language features make it unclear for students what the code actually does. They also make difficult to consolidate some important concepts, such as data typing. For example, transparent dereferencing of pointers in Ada prevents students from understanding the difference between a record and pointer to a record. Another example is automatic declaration of variables (i.e. dynamic typing) in Python. The lack of variable declaration complicates the comprehension and identification of data types and variables. Furthermore, it also muddles the concept of static scoping.

While all these features may enhance the expressiveness of a language later on, the basic concepts need to be established first in the mind of the student.

2.4 Terse and simple syntax

In the authors' opinion, the perfect candidate is a language as simple as Pascal (or even simpler), with terse syntax like C.

Pascal has been widely recognized as a good language for CS1 courses. However, its control syntax is too verbose. Also, the use of brackets and parenthesis in constructions emphasizes the formal character of the language, one source of confusion for new programmers.

In addition, Pascal syntax is more complex than needed. For example, the use of semicolons as separators instead of terminators for sentences is a problem for students. They end up guessing when to add a semicolon and when not to add one.

There is also a practical problem with Pascal. It is difficult to find an implementation of Pascal which works well in all the operating systems the students may use at home and the lab.

Ada is quite verbose and utterly complex. This makes things hard for students in introductory courses, because there are many different constructs to master and the possibility of wandering off by mistake, as explained before. Also, control structures requiring *exit when* constructs are easily misused. At the same time, this construction cannot be forbidden because it is necessary for do-while (in fact do-until) loops.

Using white space characters and tabs as part of the syntax is a double-edged sword. On one hand, it is useful to force a valid indentation (e.g. Python). On the other hand, it leads to syntax errors that are hard to solve for a first course student. For example, mixing white space characters and tabs in the same program causes errors, and it is difficult to locate them manually. Even worse, the correctness of the program depends on the text editor. Some editors hide white space characters or translate tabs to them or vice versa. A common pitfall when programming in Python is to use two different editors to write the same program (e.g., the editor installed in the laboratory and the editor installed in your personal computer). The authors consider that, in general, using these characters as part of the syntax is not desirable

in a introductory programming language.

2.5 Explicit management and debug facilities

One of the aims of a first programming course is to teach students how to debug programs.

To make memory allocation errors explicit and introduce the concept of dynamic memory (de)allocation, the language must support manual memory deallocation instead of automatic garbage collection.

In addition, it must be easy to detect dynamic memory failures and leaks. It would be also desirable to be able to inspect the program stack in a novice-friendly format without using complex tools (e.g. gdb).

2.6 Text based

The language must be suitable for a first year University course. There are several visual programming languages for education at different levels [19, 4, 11, 6, 7]. Nevertheless, the authors need a classic text-based language, closer to real world programming languages, to ease the way into the other languages taught later in the curriculum (C, Ada, Java, Python, etc.).

2.7 Editor/IDE independent

Another important requisite is that the language must be independent of IDEs. Some authors are especially critical of commercially available IDEs [11]. For the authors of Picky, it is a must to be able to compile and execute programs in the command line and from shell scripts.

In some IDEs (like Eclipse), it is very difficult for a novice to understand when a program is being compiled, when it is being ran and what version is being used. The authors experience in more advance courses on Java and Android using Eclipse is that the facilities provided for managing projects cause problems even to last year students. For example, it is quite complex for them to export and import a project, even on machines running the same operating system.

In addition, it is paramount to allow the expert users (i.e. the teachers) to select the text editor of their choice so that the class can be taught fluidly. It is very common for the authors to program something on demand as part of an explanation. If the environment is cumbersome, the students will get bored and distracted.

While syntax highlighting (or any other feature that make plain text look like formatted text) may be useful for more advanced courses, it is utterly harmful for novices. On one hand, the students are told that the compiler only accepts source code in plain text and that plain text does not have any format, it is just a sequence of character codes. On the other hand, the IDE or editor magically shows bold and italic colored fonts. The authors want to avoid this kind of *magical effects* to improve the comprehension of the tools.

2.8 Realistic I/O managing preserving referential transparency

File I/O is important not just to perform I/O, but also to teach the students how to use control structures to guide data consumption without violating file I/O rules imposed by the file abstraction.

File handling in Ada is clumsy, to put it mildly. Calling *End_Of_File* may block a program when reading from the terminal, and students will not know why. Furthermore, we teach that functions should be *referentially transparent*. Nevertheless, many Ada file I/O subprograms (that is, non-deterministic subprograms) are functions, not procedures. This violates the referential transparency.

2.9 Portable

In order to study at home and complete the assignments, students must be able to use at home the same tools that they use at the laboratory. The tools must be available for the systems they use, namely, Windows, Linux and Mac OS X. Of course, the tools must be easy to install for all these systems.

In addition, the executable files generated by the students should also be portable. The first option is to use an interpreted language. Nevertheless, interpreted languages make it difficult to consolidate concepts like compiling, linking, and executing. For the student, it is hard to distinguish between source code files and executable files. Another option is to use a compiled language that generates machine independent code to run on a virtual machine. In this case, it is also difficult to distinguish between the virtual machine and the compiled program.

The solution we have taken is to follow the latter approach and keep the illusion that the compiler generates an actual native binary file that can be executed in the system like a native executable (i.e. without invoking another program like the virtual machine).

2.10 Open source

Last, the authors need to be able to modify the tools if necessary. Thus, the language selected for the course must have open source tools available for all the systems enumerated in the previous point.

3. PICKY IS REALLY PICKY

Before providing a description of the language, we would like to summarize its main features regarding safety. As the name of the languages suggest, Picky is very restrictive. The aim is to forbid students any practice that can be harmful if it becomes a habit.

When a kid learns how to ride a bicycle it is convenient to use side-wheels for a while. Only after such artifact is under control, a new bicycle (one without side-wheels, and perhaps with an engine) is more convenient. In the same way, Picky is highly restrictive regarding what can be done and what can not in a program. It has side-wheels attached.

Apart from the desired features described before (strong typing, avoidance of *automatic* features such as dynamic declaration or automatic deferences of pointers, no global variables, and so on), both the compiler and the run time include extra checks and waste memory and time to provide additional safety features.

- If the student forgets to initialize a variable, it will not be zeroed. Moreover, the variable will not have the corresponding value left in the stack by a previous activation register.

  In Picky, all variables are implicitly initialized with a random value. Thus, if there are uninitialized variables, every execution will be different.

- The compiler does not provide *warnings*. Any error is a fatal error and the program does not compile.
- The runtime tracks dynamic memory usage and provides informative diagnostics regarding accidental use of dangling pointers.
- A program fails if there are dynamic memory leaks, i.e. if there is memory allocated and not freed before the program terminates.
- Functions do not accept parameters passed by reference.
- It is required that *return* is the last statement in the function body.
- Procedures cannot use *return*.

In addition, some constructions are forbidden. For example, the authors have detected that the following erroneous construction is very common:

```
if(condition){
       dosomething();
}else{
       ;
}
```

For the above construction, the Picky compiler gives a compilation error. This code should be rewritten as:

```
if(condition){
       dosomething();
}
```

4. THE LANGUAGE

The language is very simple. To get a full description of the language, see [2]. What follows is a discussion of the most relevant details from a pedagogical point of view, following the requirements stated in section 2. There are further pedagogical omitted here for the sake of brevity.

4.1 Programs

Picky has control structures reminiscent of C and data declarations in the style of Pascal. A source program is made of a single file. A simple hello world example:

```
1 /* Hello world  */
2
3 program Hello;
4
5 procedure main()
6 {
7       writeln("hello, world");
8 }
```

Comment syntax is taken from C. A program is introduced by a *program* clause (line 3) that assigns an identifier to the program. A procedure named *main* must be included, like in C. The program starts executing its body and terminates when returning from it.

All declarations and statements are terminated by a semicolon, but note that procedure and function definitions are not terminated by a semicolon. Constants, types, procedures, and functions may not be declared within the scope of a procedure or function. That is, subprograms may not be nested and constants and types must be declared in the global scope.

The language is case-sensitive. An identifier must start with an alphabetic character followed by zero or more alphanumeric characters. Picky only has 26 keywords and a total of 81 language defined names, including keywords, builtins and predefined constants.

A program may also include one or more constant declaration blocks, one or more type declaration blocks, one or more variable declaration blocks, and procedure and function definitions. The scope for a declaration goes from the point where it happens in the source to the end of file. Global variable declaration sections are forbidden by the compiler unless a flag is supplied.

Constant, type, and variable declaration blocks start with the keyword *consts*, *types*, and *vars* (respectively) followed by declarations. The following program is an small, correct, albeit useless, example:

```
1  program Xample;
2
3  consts:
4        Npts  = 11;
5        Greet = "hi";
6
7  types:
8        Tmonth = (Jan, Feb, Mar);
9        Tpt = record{
10             x: int;
11             y: int;
12       };
13       Tpts = array[0..Npts-1] of Tpt;
14
15 consts:
16       Zmonth = Jan;
17
18 vars:
19       a: month;
20
21 procedure incptx(ref pt: Tpt)
22 {
23       pt.x = pt.x + 1;
24 }
25
26 function addpty(p1: Tpt, p2: Tpt): Tpt
27 {
28       p1.y = p1.y + p2.y;
29       return p1;
30 }
31
32 procedure main()
33         pts: Tpts;
34         i: int;
35 {
```

```
36      for(i = 0, i < Npts){
37            pts[i] = Tpt(2, 4);
38            incptx(pts[i]);
39            pts[i] = addpty(pts[i], pts[0]);
40      }
41      writeln(pts[Npts-1].x);
42      writeln(Greet);
43 }
```

4.2 Basic data types

Per requirement 2.3, Picky is strongly typed. The basic types are *bool*, *char*, *int*, *float*, and *file*. They correspond to booleans, characters, integers, real numbers in floating point, and external (text) files.

Two types are compatible (for assignment and other operators) only if they have the same name. Predefined types also obey the same rule. Constants and literals are an exception, they belong to *universal* types that are assumed to be compatible with any basic data type of the same kind. This is reasonable, for example, to permit using integer literals in expressions that belong to a user defined integer type. Another exception are subranges. Subranges do not introduce a new type; they declare a restriction defining a subset of an existing type.

A type definition defines a new type and declares its name. For example:

```
types:
      Apples = int;
      Oranges = int;
```

This code defines two new types: *Apples* and *Oranges*. It is not legal to mix apples with oranges, and it is not legal to mix any of them with *int* values. However, integer constants and literals may be mixed with any of them.

In general, the language does not permit type casts. However, type casts are permitted to convert ordinals to the integer representing their position in the type and vice versa. Also, integers may be converted to floating point numbers and vice versa.

To convert a value to a type use the target type name as a function. For example, these are legal expressions:

```
char(int('A') + 1)
float(3)
int(4.2)
```

4.3 Explicit dynamic memory and resource management

Resources in Picky are managed explicitly as stated in section 2.5. Memory allocations and deallocations are explicit and there is no garbage collection.

A pointer data type refers to another type and permits using *new* and *dispose* to handle dynamic variables of the pointed-to type. Type definition uses the ^ notation, taken from Pascal:

```
types:
      Arry = array[1..10] of int;
      Iptr =  ^int;
      Aptr = ^Arry;
```

The second line declares an array data type used in the last line, to declare a *pointer to Array* data type. The third line declares a pointer to integer. It is legal to declare a pointer to a type that is not yet defined in the program, but the target type must de defined later. This permits declaring circular data types, like linked lists. In no other case may a type be defined in terms of not yet defined types.

Syntax to dereference a pointer value is also taken from Pascal, and also uses the ^ sign:

```
iptr^ = 2;
aptr^[1] = iptr^;
```

L-values of pointer types may use the following procedures to allocate and deallocate memory: *new*(*ptr*) (set ptr to point to newly allocated memory) and *dispose*(*ptr*) (frees the memory referenced by *ptr*). All memory allocated with *new* must be released by calling *dispose* before completion of the program, or the program will abort and report memory leaks. The interpreter makes sure that dereferencing a dangling pointer (i.e. a pointer pointing to freed memory) will abort the execution, providing the corresponding error to the user.

File descriptors are also managed explicitly. Files need to be opened and closed using the appropriate builtins, *open*(*file*) and *close*(*file*). Any error related to accessing a file is fatal for the program.

4.4 Input/Output

Some languages use I/O primitives that are predictable but too low-level. Others provide high-level, but unpredictable facilities. Among other things, it is impossible, in general, to know if there is an *end of file* before trying to read. On the other hand, it is not reasonable to read without checking the *end of file* condition.

As we explain to our students in the CS1 course, when programming, side effects must be contained. Checking for the *end of file* should be a function without side effects. The *read* operation should be a procedure with side effects.

In Picky the I/O primitives follow the requirements stated in section 2.8. They are both practical and clean from a theoretical point of view. A *peek* procedure scans the input to check for *end of file* or *end of line* conditions. Part of the *peek* specification is that it may read internally from the file. The *eof* operation is a function and has no side-effects (i.e. it never reads). Before any attempt to call *read* or *peek*, *eof* returns false as it should.

The language forbids to read *end of line* marks, they must be skipped. The runtime includes checks to trigger errors if a program tries to read them directly instead of using a *readeol* primitive.

4.5 Debugging facilities

Following the requirement in section 2.5, built-in procedures are provided for user friendly debugging, and abnormal termination: *fatal(text)* (print text and abort execution), *stack()* (dump the stack in a friendly format for debugging) and *data()* (dump global data in a friendly format for debugging). For example:

```
stack trace at:
dowork() pid 0 pc 0x000008 xample.p:9
arguments:
x = 3
local variables:
z = 8

called from:
main() pid 1 pc 0x000016 xample.p:16
local variables:
x = 3
```

In other development environments, students tend to debug by using step-by-step execution on debuggers instead of thinking. In this language, it is natural for them to dump the program state and think about the cause of their problems. Later, when they are less prone to misuse them, they will learn more advanced debugging techniques such as step-by-step execution, breakpoints, etc.

4.6 Procedures and functions

There is a clear separation in Picky between procedures and functions to follow the principle described in 2.8. The principle is that functions should have no lateral effects and should preserve referential transparency. This principle is also followed by builtin functions and procedures. Procedures are named actions, so can have lateral effects, and do not return values. Argument passing is by value (by default) or by reference (using the keyword *ref* before an argument name). See lines 21-24 in the *Xample* program. Functions are declared similarly, see lines 26-30 in the program.

4.7 Global and local variables

Picky does not permit global variables by default. They can be enabled with a compiler flag. The flag is in place so that the concept of global variables can be explained in the corresponding class.

It is not allowed to declare a type on the fly in the variable declaration, unlike in Pascal. A type identifier is required after the colon. This forces the students to define types first and assign them meaningful names before using them.

Variables are initialized to random values. This feature makes programs fail when using uninitialized variables instead of making them work intermittently. Therefore, students learn quickly that uninitialized variables are dangerous.

4.8 Control structures

Picky has the usual control structures. The *if*, *while*, *do-while*, and *switch* statements borrow their syntax from C and semantics from Pascal (there is no *break*). Statements used for *then* and *else* arms must always be blocks. Students face no *dangling else* in Picky.

The *for* loop (see lines 36-40 in *Xample*) has a header with only two expressions, an initialization and a condition. The initialization must be an assignment for a variable of an ordinal type. The condition must use any of these operators: "<", "<=", ">" or ">=". The first two ones make the variable increase automatically after each iteration. The last two make the variable decrease automatically after each iteration.

After the *for* loop, the control variable is equal to the value on the right of the condition. This implies that there is no out of range condition for the control variable even when using "<=" or ">=" with the first or last valid value of an ordinal type. In *Xample*, *i* value is *Npts* when the loop is done.

The only way to exit a loop is to satisfy the condition of the loop; there is no *break* or *goto* statement. This way the postconditions are clear and the student is forced to structure the program.

5. COMPILATION AND EXECUTION

The Picky compiler, *pick*, is implemented in C. The compiler is implemented using *yacc* [13] and should be easy to understand.

The compiler does not emit warnings. All diagnostics correspond to compile time errors. In many cases, when an error is detected, a symbol or node in the syntax tree is still built, for safety; other parts of the compiler still get a data structure as expected, and it's less likely that an invalid value causes a bug.

Picky compiles to a virtual machine (PAM [2]), invoked transparently by the compiled output file. Thus, students are not surprised by "binaries" behaving differently on different platforms. Code generation is straightforward. The machine is stack based. Most operations take arguments from the stack and replace them with a result, pushed also on the stack. There is a single flow of control, guided by a loop switching on the instruction type.

PAM wastes memory and time to detect mistakes like out of range conditions, the use of already disposed data structures, etc. This way, it issues very descriptive diagnostics and not just "*segmentation violation*".

As already stated, variables (from the data, stack or heap) are initialized with random values, to let the user discover early that variable initialization is missing. Such random values are always odd, to recognize uninitialized pointer values and issue a descriptive diagnostic for that case at run time, instead of a *segmentation violation* or producing a heisenbug.

The abstract machine construction makes it possible to dump the state at any point in a user friendly format. The *stack* and *data* builtins (explained in section 4.5) rely on this feature.

Picky "binaries" are just text files that are interpreted by PAM. They start with the Unix *hash bang* syntax to call PAM on their own. In Windows, to the same end, the file

extension *pam* is associated in the registry to the application *pam* as part of the installation. Thus, students have "binary" files that, at the same time, are portable and can be used for pedagogical purposes. Students compile and then run the resulting file:

```
prompt$  pick hello.p
prompt$  out.pam
 hello picky!
prompt$
```

The "binary" generated includes portions of the source code in comments, and can be used during lectures to teach how the code written by students maps to machine instructions:

```
#!/bin/pam
entry 0
...
# x: int = 3
0000a push 0x00000003 # 3;
0000c lvar 0x00000000 # x;
0000e sto 0x00000002
# dowork(x: int)
00010 lvar 0x00000000 # x;
00012 ind 0x00000004
00014 call 0x00000000 # dowork();
```

This way students do not perceive the machine as a *magical device*.

6. EXPERIENCE

The authors are quite happy with the results of using Picky in CS1 courses. They have used the language to teach nine CS1 courses that are part of three different degrees of the Telecommunications Engineering School of the Rey Juan Carlos University of Madrid. The number of students that have actually used the language is greater than six hundred. The first generation of students that used Picky for CS1 is currently programming in Java, C, Ada, Python and shell scripting in 3rd-year courses.

It is difficult to evaluate fairly and accurately the effectiveness of the language for teaching CS1 courses. Since the authors are in charge of teaching and evaluating the students, any evidence related to grades of tests and assignments could be unintentionally biased. In addition, given the continuous turmoil of secondary education in Spain, which creates a high heterogeneity of students at different points in time it is difficult to quantify any approach.

In order to assess some feedback from the students, we passed a survey in a 3rd-year course class. Of course, this survey should not be considered an indisputable evidence, but it points in the right direction. We polled 3rd-year students because they have learnt other programming languages and have a wider vision. On the other hand, there is an implicit bias because many students abandon the degree (for many reasons, but the common case is the difficulty of the degree, not necessarily CS1). The results of the survey are shown in Figure 1. The questions were:

(A) *How did you like Picky as your first language programming language?*

(B) *Did using a simple language in CS1 helped you to learn more than a complex but powerful language?*

(C) *Was it difficult to learn the Picky syntax in CS1?*

(D) *Was it difficult to learn the Ada syntax in CS1?*

Questions A to C were given to students that used Picky as a first language. Question D was given to students that used Ada instead.

The experience with the language is positive. We do see the students less engaged in nitpicking with the unimportant details of the language and more focused on the learning task. In our opinion, Picky has made teaching simpler and the students learn more compared to other introductory courses the authors have taught in Ada and C. Before using Picky, the authors had to explain to students how things in practice departed from what was taught in theory. This was an imposition of the language being used (e.g. the *eof* function with side effects in Ada). In addition, the students had problems regarding dynamic memory, uninitialized memory, and all the other issues enumerated early in this paper. Picky has alleviated most of these problems.

One disadvantage of creating a custom language for the course, is the absence of ready-made materials for teaching the subject and for student consultation. In order to cover this gap, we wrote an introductory programming book (in Spanish) using Picky [3] for the course. This book covers the course following the same approach and in the same order we cover it in class. It serves two purposes. On the one hand, it is a reference material for the students, with some extra content for the more advanced students. On the other hand it serves as a guide for the teachers, helping to provide a detailed guideline of what should be taught in class and in what order.

The absence of ready-made code snippets to copy from the network helps make the students work more in their assignments and spend less time forcing code copied from a random web page into them.

Another unanticipated benefit of using a language built by ourselves, is, of course, that we understand it thoroughly. With more complicated languages, it is always possible to have a dark corner of the language appear in code written by students which puzzles the teacher, sometimes momentarily, sometimes longer. While the response to the student is simple: "rewrite that mess", more advanced students may want to understand what exactly is going on. For instance, one of the authors remembers fondly trying to understand an accidental and obscure variation on the Duff device [9] to be able to explain to a good student why his code worked. With Picky, these days are over.

As every teacher knows, plagiarism detection is an important issue whenever students are given assignments. While we were concerned when we started that we would have to write our own tools for this purpose, we found that the already existing tool Moss [1] works very well with Picky and we use it routinely on the assignments.

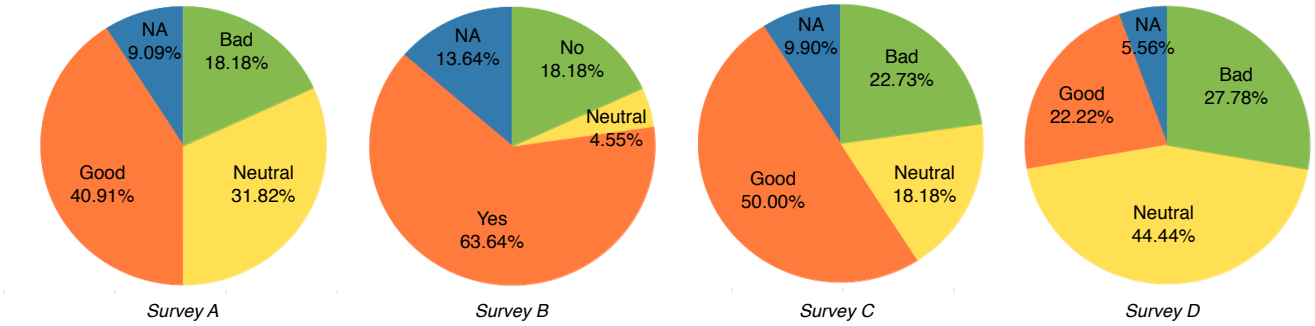

Figure 1: Students' response to Survey Questions.

7. CONCLUSIONS

The authors have designed and implemented a new language, Picky, specifically designed to aid students as much as possible during introductory programming courses. Just its I/O design would justify its adoption in the authors' opinion, but it helps students and teachers in many other ways. The language has been ported to all the mainstream operating systems. The authors have also written a book based on it for the course. While it may seem to be too much work, the return on investment is very good. The students learn more, the teaching task is simpler and the teachers have well honed tools designed exactly for the task at hand. The main regret of the authors is not having done it earlier.

8. FUTURE WORK

The authors have observed that the students are more engaged with programming when they perceive than they can program any kind of graphical game, however simple. A simple experiment with a text based animation (which was quite contrived to write in Picky), got them very interested and peaked their motivation. This insight has led us to add some builtins for simple graphic interaction in Picky (simple non-blocking keystroke I/O, sleeping and some graphical primitives). This is an ongoing effort and it is still too early to know what will come out of it.

The authors have also developed another fully compatible version of the compiler in the Go programming language [10], which may help develop new tools for use in the lab.

9. ACKNOWLEDGMENTS

This work has been supported in part by Spanish Ministry of Education project TIN2013-47030-P and Comunidad de Madrid project S2013/ICE-2894.

10. REFERENCES

[1] A. Aiken et al. Moss: A system for detecting software plagiarism. `http://theory.stanford.edu/~aiken/moss/`, 2014. Accessed: 2015-06-06.

[2] F. J. Ballesteros. *Picky Whitepaper*. lsub.org, 2012.

[3] F. J. Ballesteros, E. Soriano-Salvador, and G. G. Múzquiz. *Fundamentos de la programación*. lsub.org, 2012.

[4] M. Boshernitsan and M. S. Downes. *Visual programming languages: A survey*. Citeseer, 2004.

[5] M. Butler, M. Morgan, et al. Learning challenges faced by novice programming students studying high level and low feedback concepts. In *Proceedings of Ascilite*, pages 99–107. Citeseer, 2007.

[6] B. A. Calloni and D. J. Bagert. Iconic programming in baccii vs. textual programming: which is a better learning environment? *SIGCSE Bull.*, 26(1):188–192, Mar. 1994.

[7] T. Crews and C. Murphy. *A Guide to Working With Visual Logic*. Course Technology Press, Boston, MA, United States, 1 edition, 2008.

[8] R. Decker and S. Hirshfield. Top-down teaching: object-oriented programming in cs 1. In *ACM SIGCSE Bulletin*, volume 25, pages 270–273. ACM, 1993.

[9] T. Duff. Tom duff on duff's device. `http://www.lysator.liu.se/c/duffs-device.html`, 2015. Accessed: 2015-06-06.

[10] Google. The go programming language. `http://golang.org`, 2014. Accessed: 2015-06-06.

[11] J. H. Greyling, C. Cilliers, and A. Calitz. B#: The development and assessment of an iconic programming tool for novice programmers. In *Information Technology Based Higher Education and Training, 2006. ITHET '06. 7th International Conference on*, pages 367–375, 2006.

[12] S. Hadjerrouit. Java as first programming language: a critical evaluation. *SIGCSE Bull.*, 30(2):43–47, June 1998.

[13] S. C. Johnson. Yacc: yet another compiler-compiler. *Bell Laboratories*, 1986.

[14] D. Krpan and I. Bilobrk. Introductory programming languages in higher education. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 1331–1336, 2011.

[15] R. Lister, A. Berglund, T. Clear, J. Bergin, K. Garvin-Doxas, B. Hanks, L. Hitchner, A. Luxton-Reilly, K. Sanders, C. Schulte, and J. L. Whalley. Research perspectives on the objects-early debate. *SIGCSE Bull.*, 38(4):146–165, June 2006.

[16] R. Mody. C in education and software engineering. *ACM SIGCSE Bulletin*, 23(3):45–56, 1991.

[17] J. Ophel. Incorporating an object-oriented programming language into the first year of a software engineering education. In *Software Engineering: Education and Practice, 1996. Proceedings.*

International Conference, pages 317–322, 1996.

[18] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39(4):204–223, Dec. 2007.

[19] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[20] E. S. Roberts. Using c in cs1: evaluating the stanford experience. In *ACM SIGCSE Bulletin*, volume 25, pages 117–121. ACM, 1993.

[21] R. M. Siegfried, D. Chays, and K. Herbert. Will there ever be consensus on cs1? In H. R. Arabnia, V. A. Clincy, and N. Tadayon, editors, *Proceedings of the 2008 International Conference on Frontiers in Education, FECS 2008, July 14-17, 2008, Las Vegas, Nevada, USA*, pages 18–23. CSREA Press, 2008.

[22] M. Vujoševic-Janicic and D. Tošic. The role of programming paradigms in the first programming courses. *The Teaching of Mathematics*, 11(2):63–83, 2008.